TECHNISCHE UNIVERSITÄT WIEN
Vienna | Austria

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

S&P SECURITY & PRIVACY GROUP

Foundations and tools for the static analysis of Ethereum smart contracts

Matteo Maffei and Clara Schneidewind

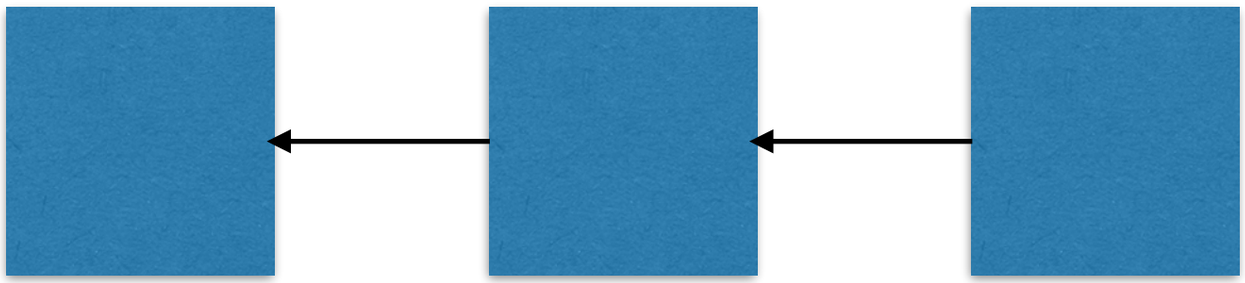
Outline

Introduction to Ethereum

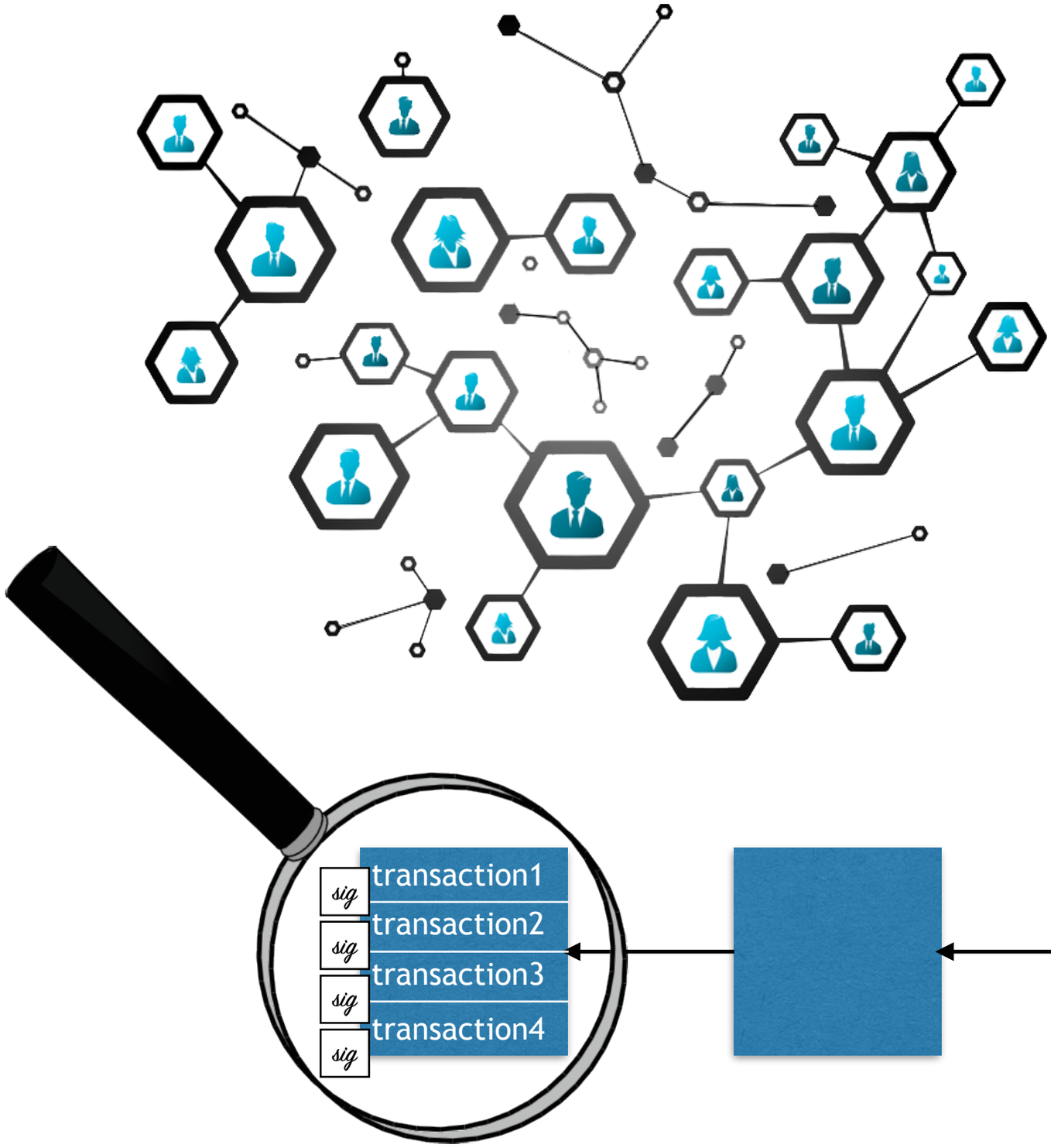
Semantics of EVM bytecode

Static Analysis of EVM bytecode

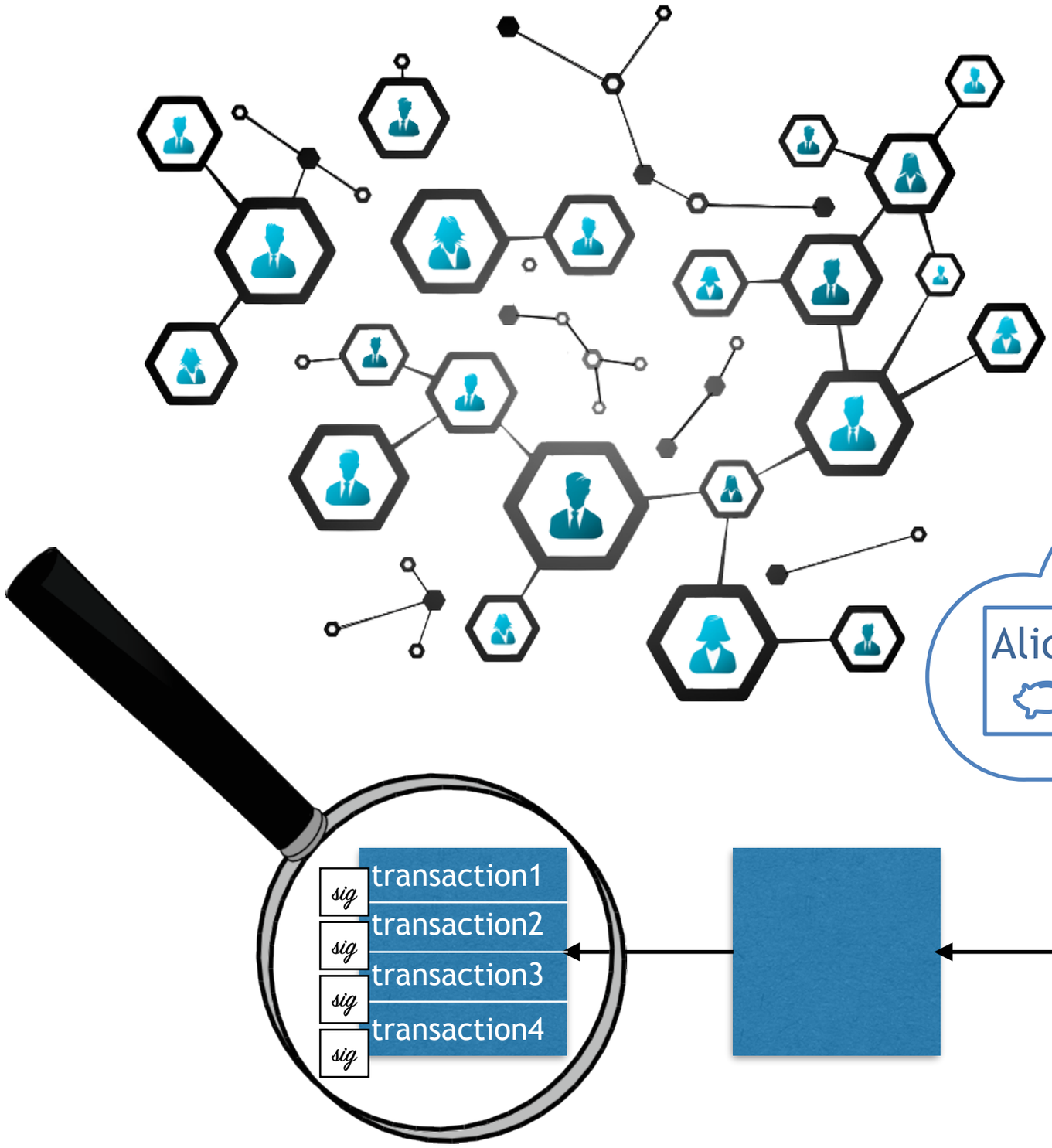
Blockchain - an overview



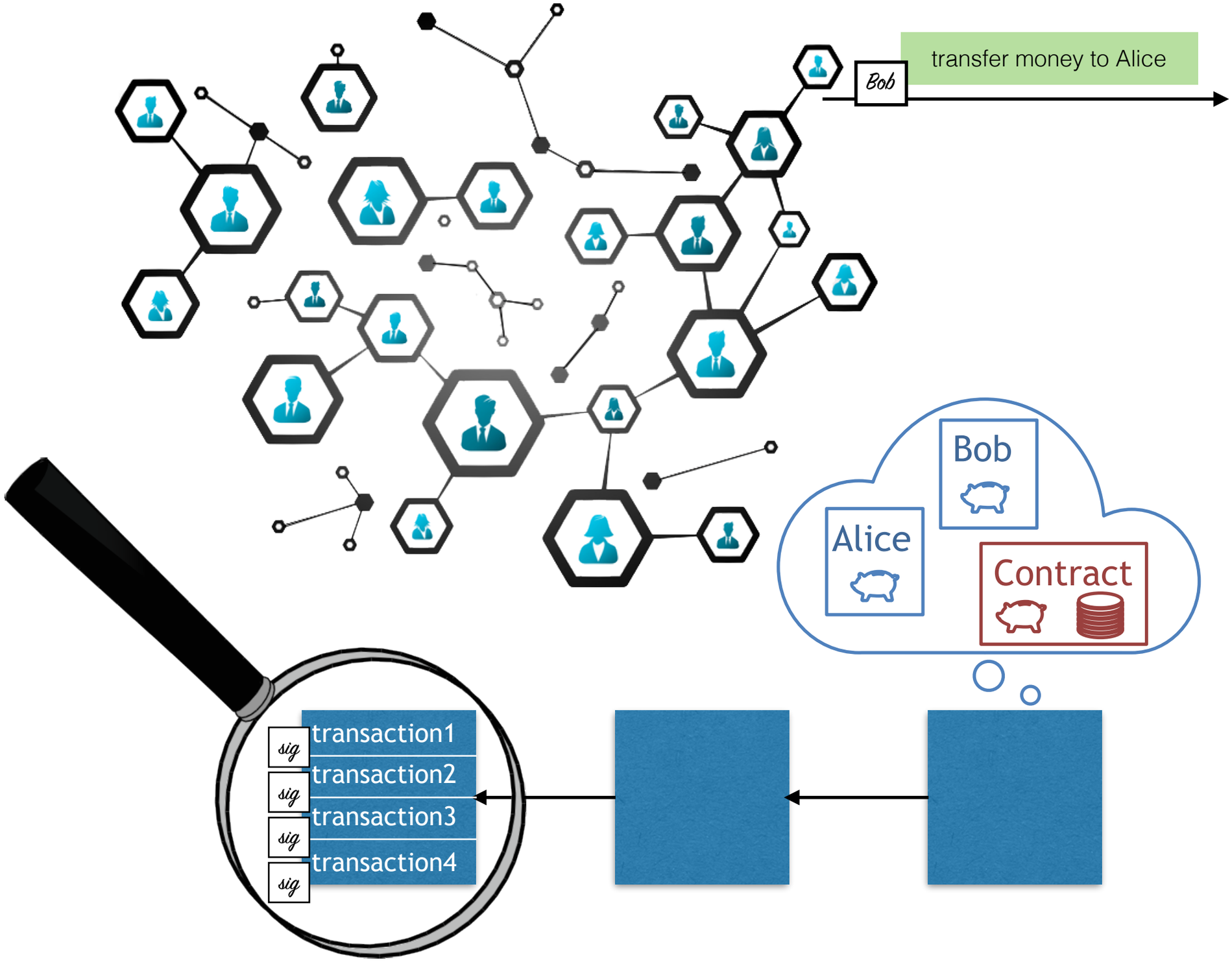
Blockchain - an overview



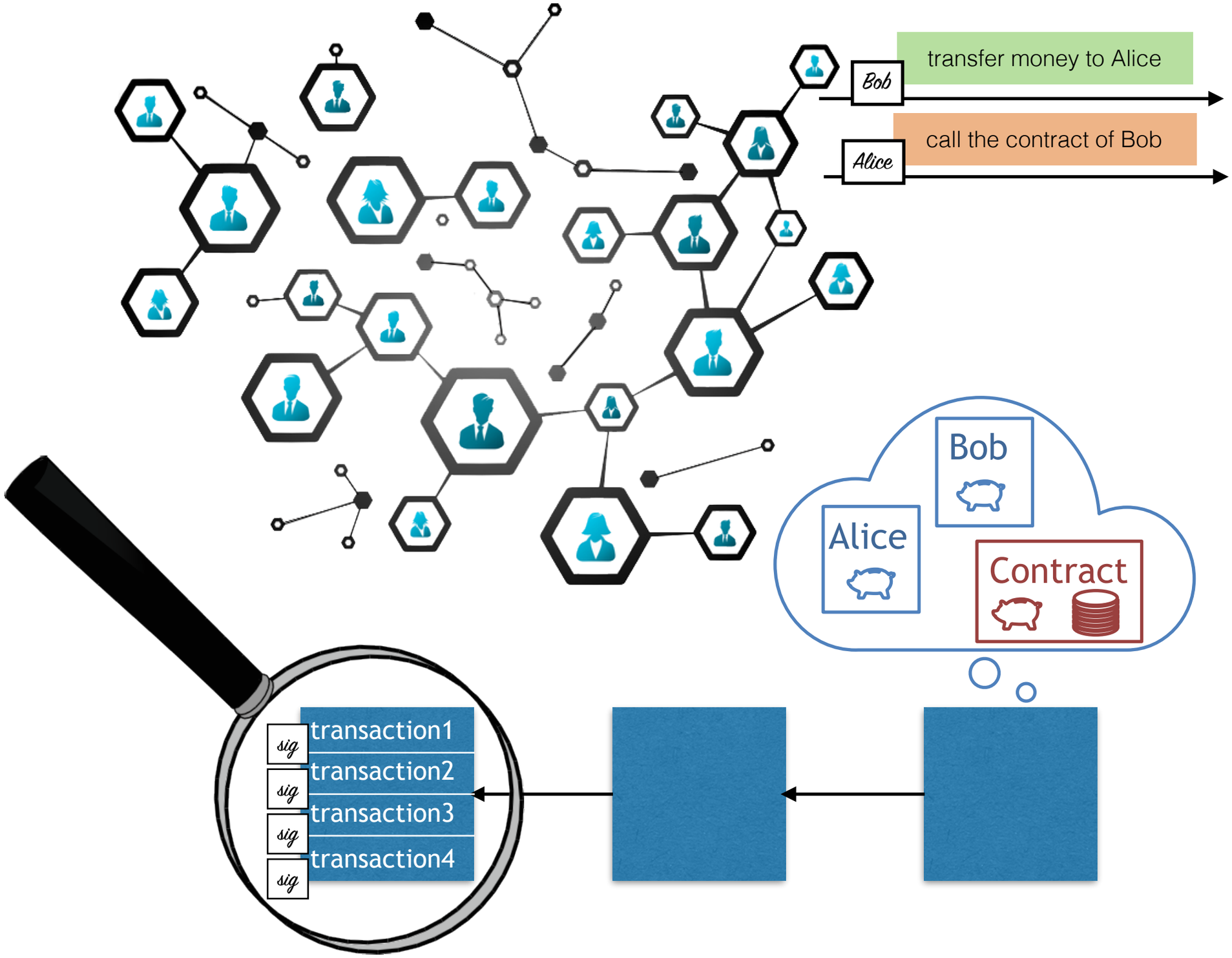
Blockchain – an overview



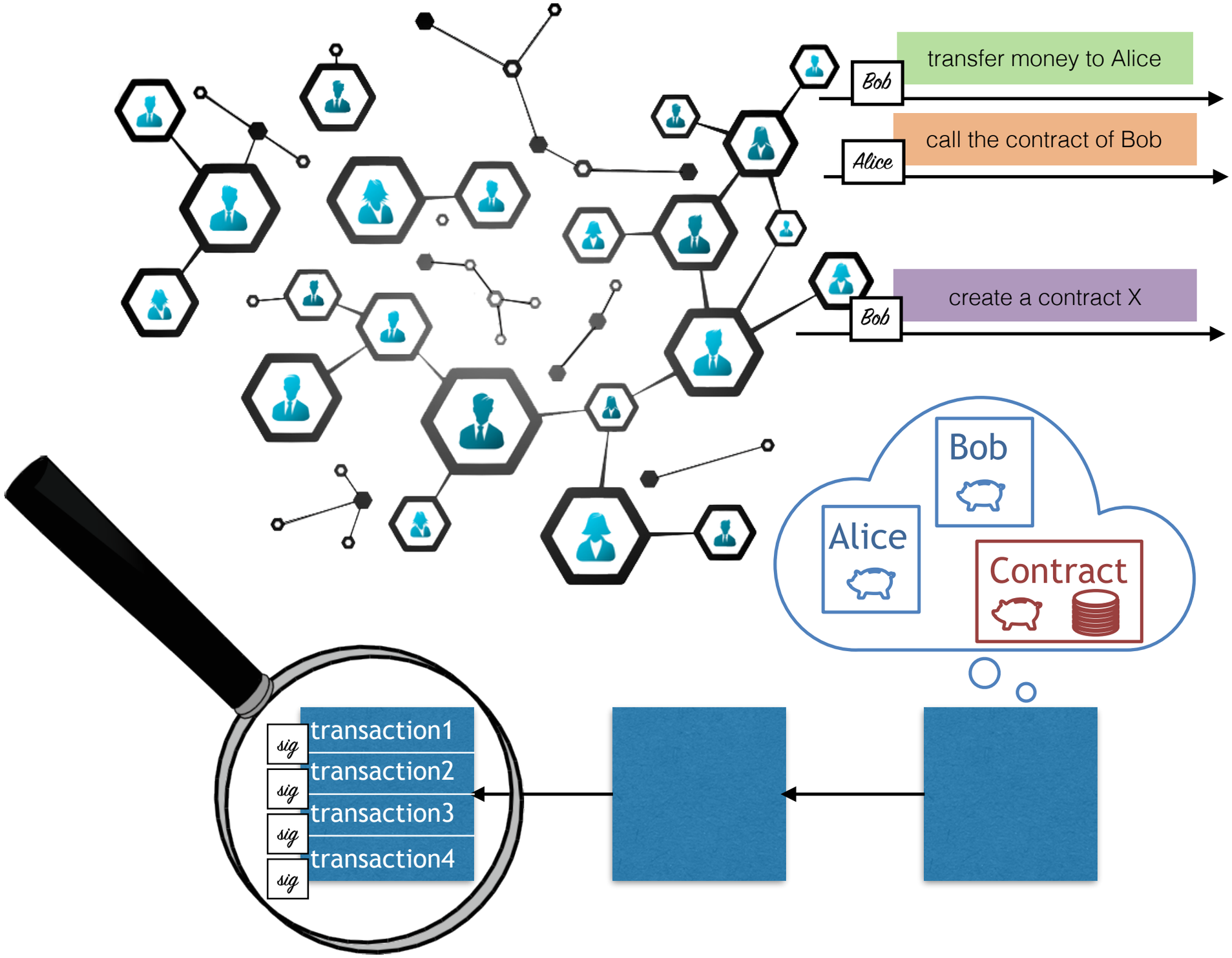
Blockchain – an overview



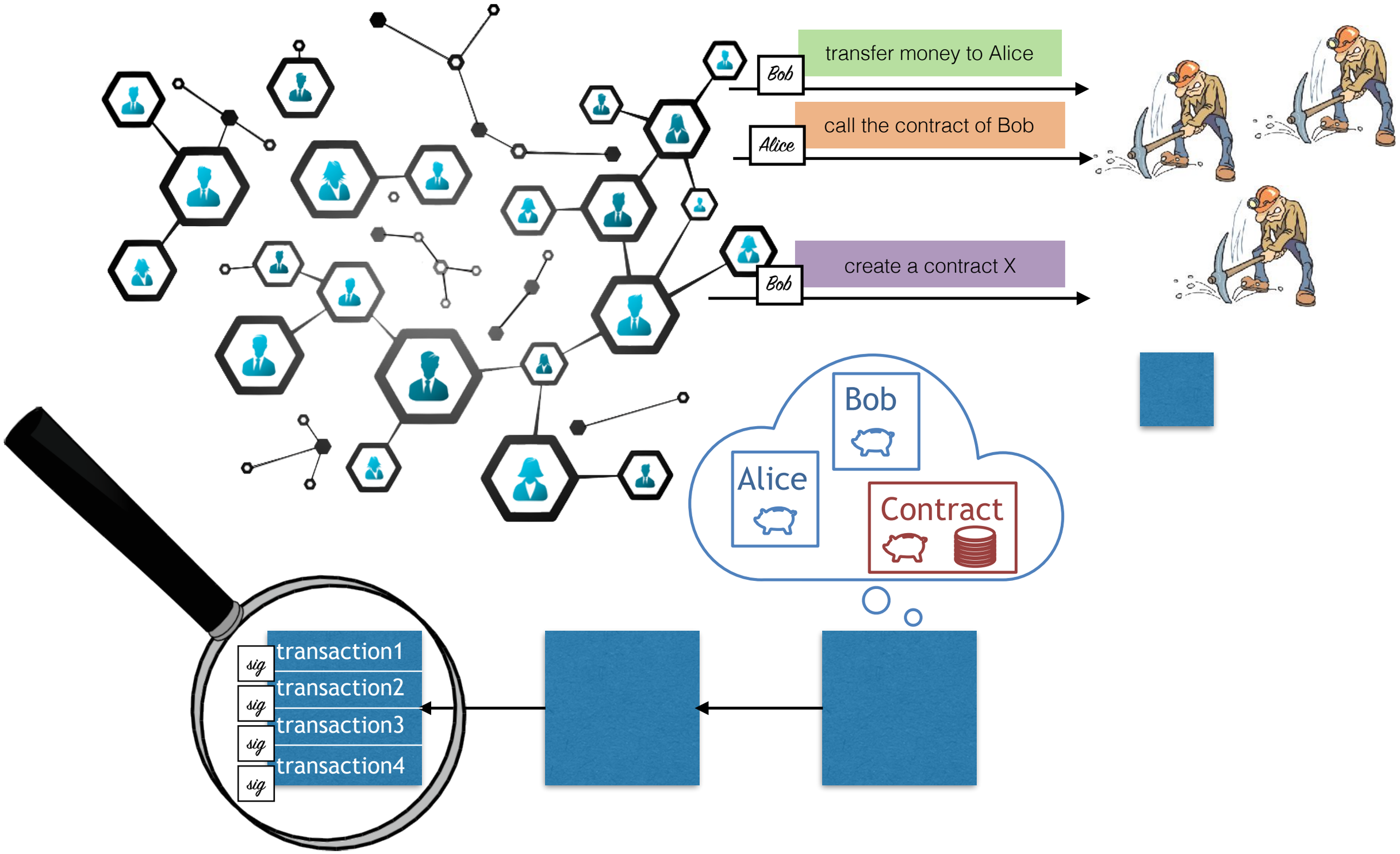
Blockchain – an overview



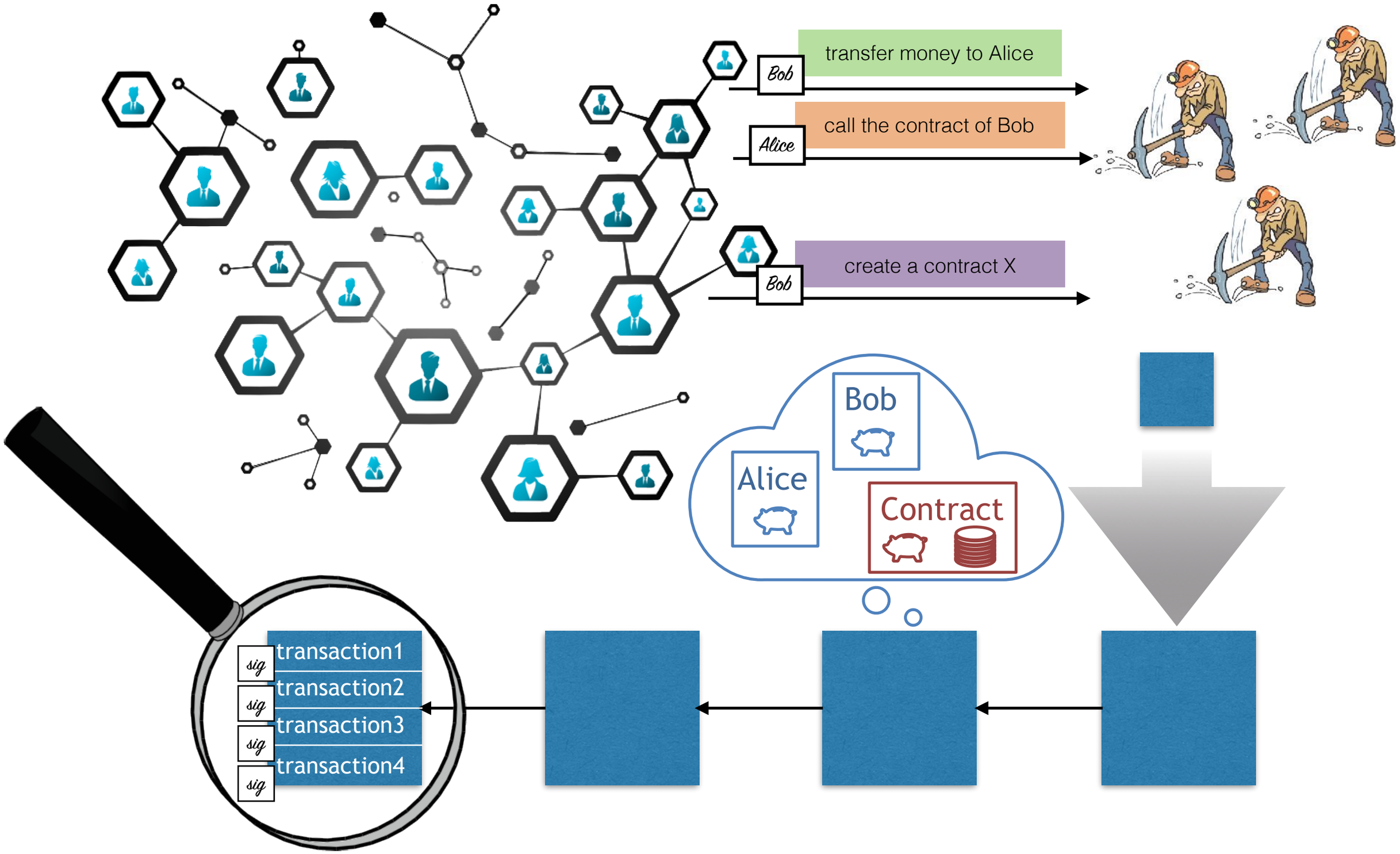
Blockchain – an overview



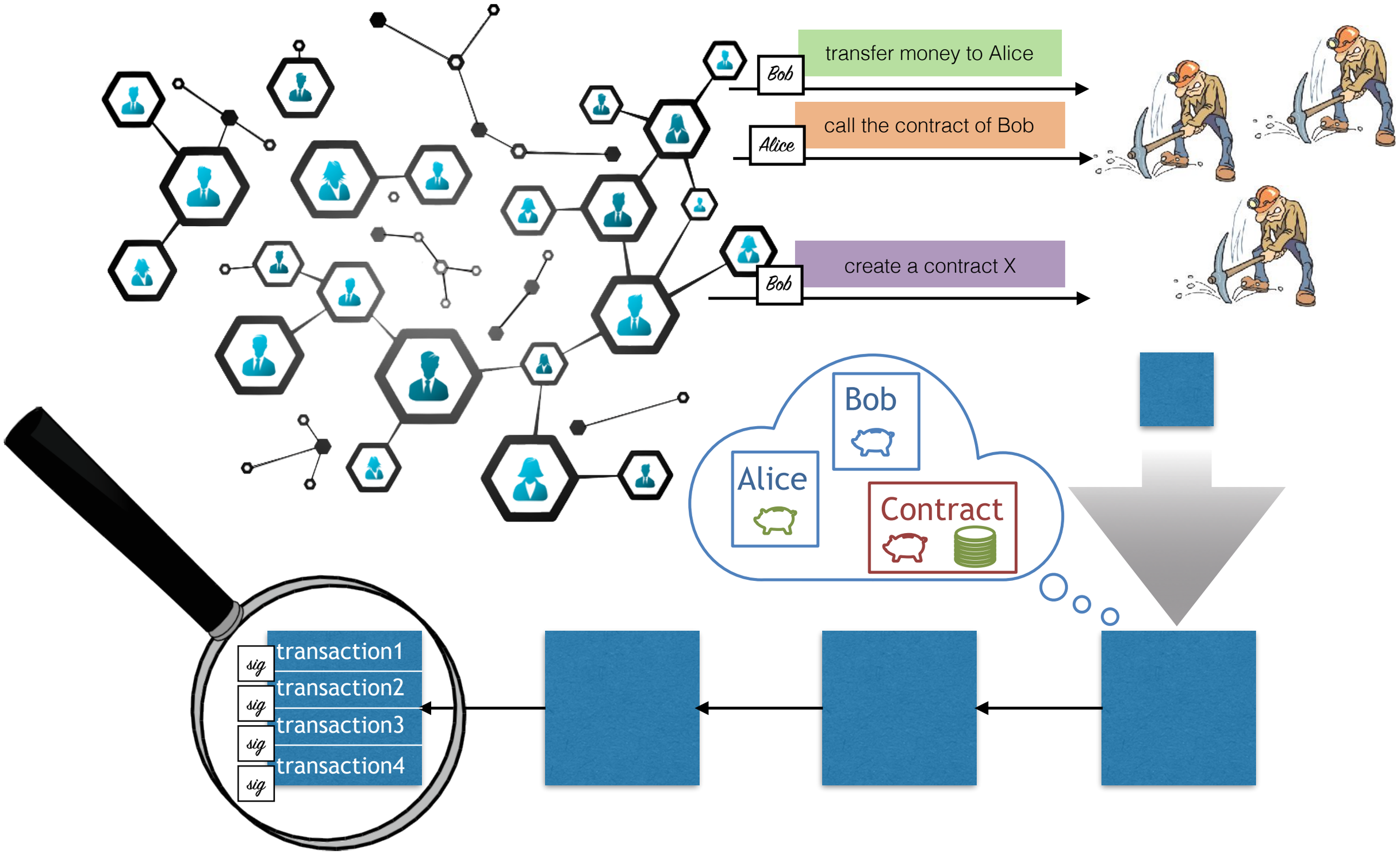
Blockchain – an overview



Blockchain – an overview



Blockchain – an overview

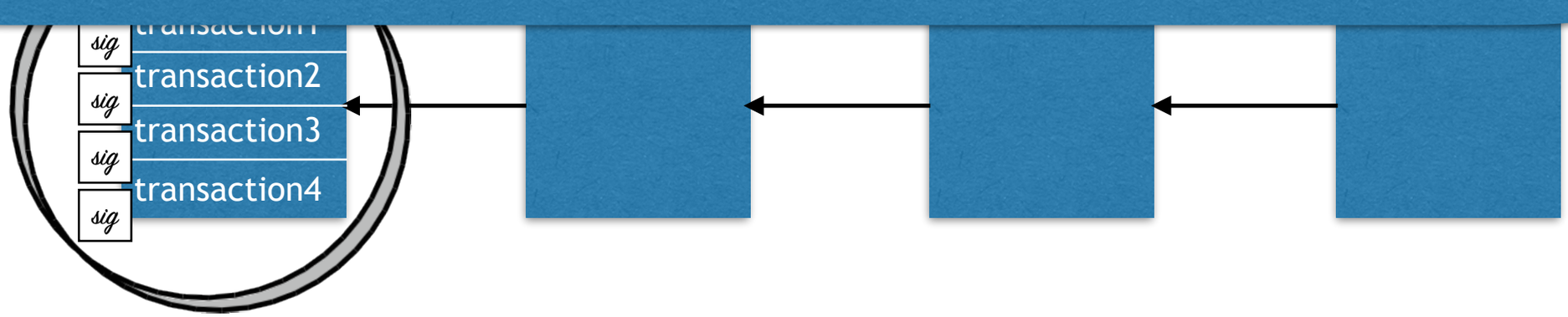


Blockchain – an overview

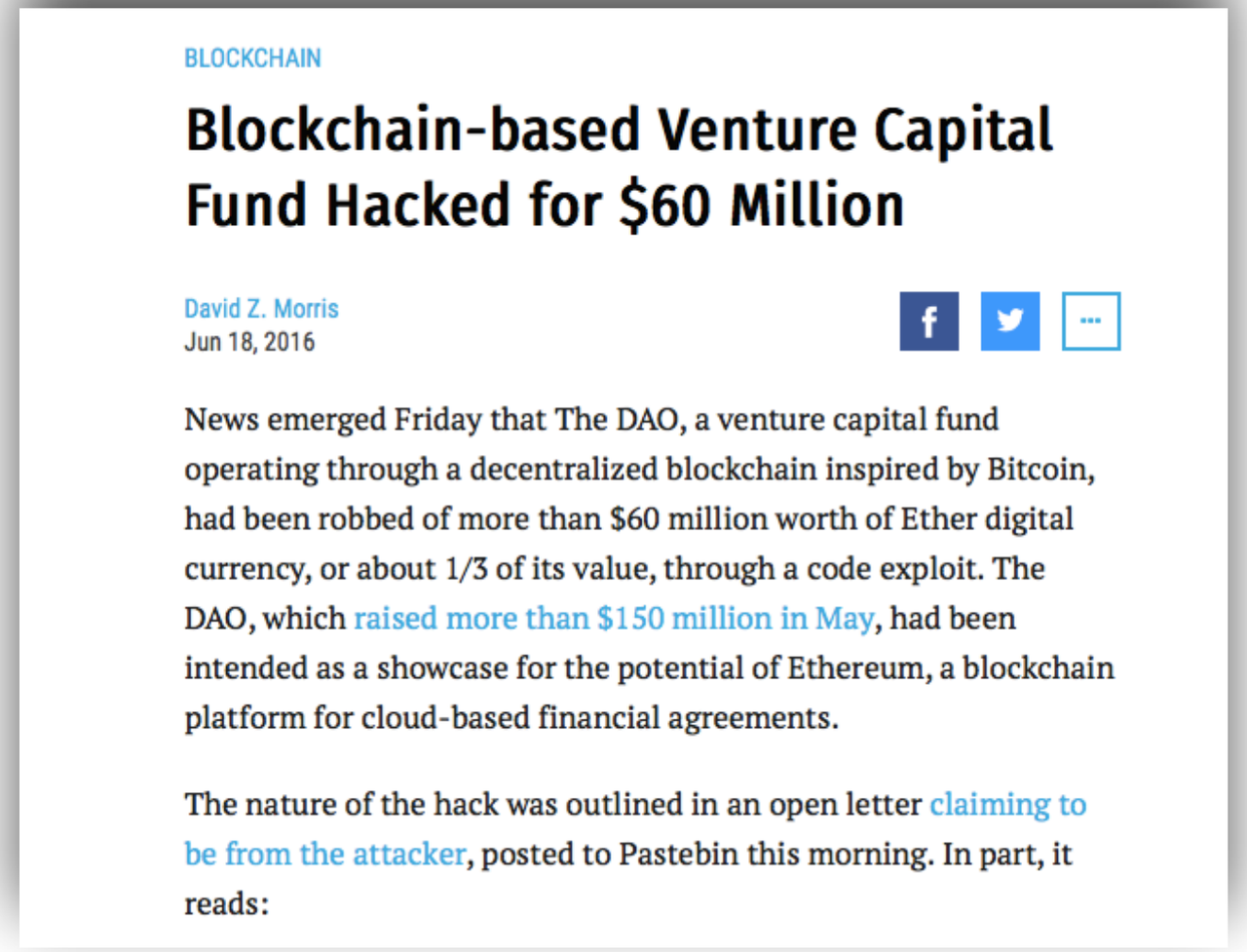
Exciting field for researchers and practitioners

Three layer architecture: programs, consensus, network

All of that works by a fascinating combination of game theory, probabilistic consensus, cryptography, and *programming language semantics*



Motivation



BLOCKCHAIN

Blockchain-based Venture Capital Fund Hacked for $60 Million

David Z. Morris
Jun 18, 2016

News emerged Friday that The DAO, a venture capital fund operating through a decentralized blockchain inspired by Bitcoin, had been robbed of more than $60 million worth of Ether digital currency, or about 1/3 of its value, through a code exploit. The DAO, which raised more than $150 million in May, had been intended as a showcase for the potential of Ethereum, a blockchain platform for cloud-based financial agreements.

The nature of the hack was outlined in an open letter claiming to be from the attacker, posted to Pastebin this morning. In part, it reads:

Motivation

BLOCKCHAIN

Blockchain-based Venture Capital Fund Hacked for $60 Million

David Z. Morris
Jun 18, 2016

News emerge
operating th
had been rob
currency, or
DAO, which
intended as
platform for

The nature o
be from the
reads:

coindesk Blockchain 101 Technology Markets Business Data & Research Consensus

Tickets are selling fast. Register for Consensus today!

Parity Team Publishes Postmortem on $160 Million Ether Freeze

Motivation

Parity Multisig Hacked. Again

Yesterday, Parity Multisig Wallet was hacked again:
https://paritytech.io/blog/security-alert.html

> *"This means that currently no funds can be moved out of the [ANY Parity] multi-sig wallets"*

A lot of people/companies/ICOs are using Parity-generated multisig wallets.
About $300M is frozen and (probably) lost forever.

Disclaimer: I lost little money (about $1000) but my friends lost about $300K.

reads:

Motivation

Parity Multisig Hacked. Again

Yesterday, Parity Multisig Wallet was hacked again:
https://paritytech.io/blog/security-alert.html

> *"This means that currently no funds can be moved out of the sig wallets"*

A lot of people/companies/ICOs are using Parity-generate
About $300M is frozen and (probably) lost forever.

Disclaimer: I lost little money (about $1000) but my frien

reads:

A survey of attacks on Ethereum smart contracts

Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli

Università degli Studi di Cagliari, Cagliari, Italy
{atzeinicola,bart,t.cimoli}@unica.it

Abstract. Smart contracts are computer programs that can be correctly executed by a network of mutually distrusting nodes, without the need of an external trusted authority. Since smart contracts handle and transfer assets of considerable value, besides their correct execution it is also crucial that their implementation is secure against attacks which aim at stealing or tampering the assets. We study this problem in Ethereum, the most well-known and used framework for smart contracts so far. We analyse the security vulnerabilities of Ethereum smart contracts, providing a taxonomy of common programming pitfalls which may lead to vulnerabilities. We show a series of attacks which exploit these vulnerabilities, allowing an adversary to steal money or cause other damage.

1 Introduction

The success of Bitcoin, a decentralised cryptographic currency that reached a capitalisation of 10 billions of dollars since its launch in 2009, has raised considerable interest both in industry and in academia. Industries — as well as national governments [48,55] — are attracted by the "disruptive" potential of the *blockchain*, the underlying technology of cryptocurrencies. Basically, a blockchain is an append-only data structure maintained by the nodes of a peer-to-peer network. Cryptocurrencies use the blockchain as a public ledger where they record all the transfers of currency, in order to avoid double-spending of money.

Although Bitcoin is the most paradigmatic application of blockchain technologies, there are other applications far beyond cryptocurrencies: e.g., financial products and services, tracking the ownership of various kinds of properties, digital identity verification, voting, *etc.* A hot topic is how to leverage on blockchain technologies to implement *smart contracts* [34,54]. Very abstractly, smart contracts are agreements between mutually distrusting participants, which are automatically enforced by the consensus mechanism of the blockchain — without relying on a trusted authority.

The most prominent framework for smart contracts is Ethereum [32], whose capitalisation has reached 1 billion dollars since its launch in July 2015[1]. In Ethereum, smart contracts are rendered as computer programs, written in a Turing-complete language. The consensus protocol of Ethereum, which specifies how the nodes of the peer-to-peer network extend the blockchain, has the goal

[1] https://coinmarketcap.com/currencies/ethereum

Smart Contracts

- Typically written in Solidity (weird JavaScript variant)
- New languages are emerging (weird Python variant)

```
contract SimpleStorage {
    uint storedData;

    function set(uint x) {
        storedData = x;
    }

    function get() constant returns (uint retVal) {
        return storedData;
    }
}
```

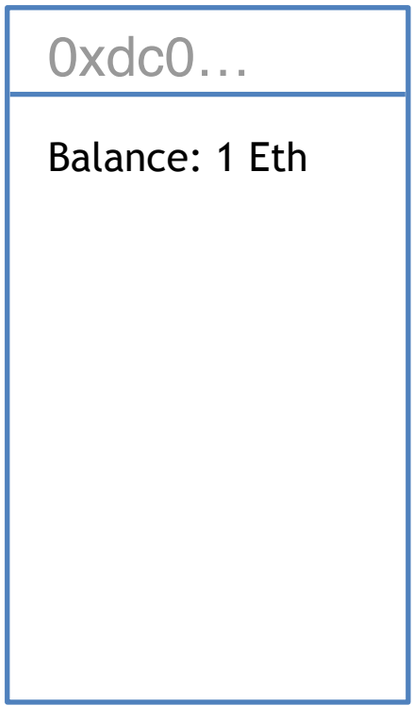
- Uploaded on the blockchain as EVM bytecode

```
PUSH1 0x01
PUSH1 0x60
MSTORE
PUSH1 0x20
PUSH1 0x40
PUSH1 0x01
PUSH1 0x60
PUSH1 0x00
PUSH32 0x0318247CB34f134f3cF49E97647227dc2D75Abe8
GAS
CALL
```

Overview on Ethereum

Overview on Ethereum

External Account



Overview on Ethereum

External Account

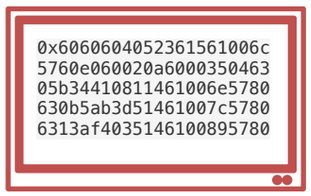
0xdc0…

Balance: 1 Eth

Contract Account

0xde2…

Balance: 1 Eth



Overview on Ethereum

External Account

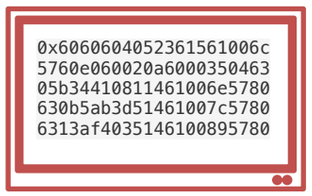
0xdc0…

Balance: 2 Eth

Contract Account

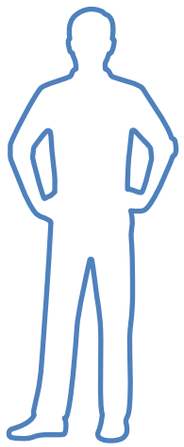
0xde2…

Balance: 1 Eth



Transfer

1Eth



External Account

0xa30…

Balance: 4 Eth

Overview on Ethereum

External Account

0xdc0…

Balance: 2 Eth

Contract Account

0xde2…

Balance: 1 Eth

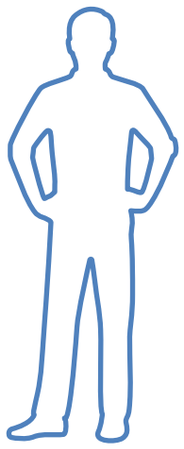
Contract Account

0xfe0…

Balance: 3 Eth

Transfer

1Eth

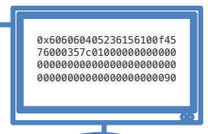


External Account

0xa30…

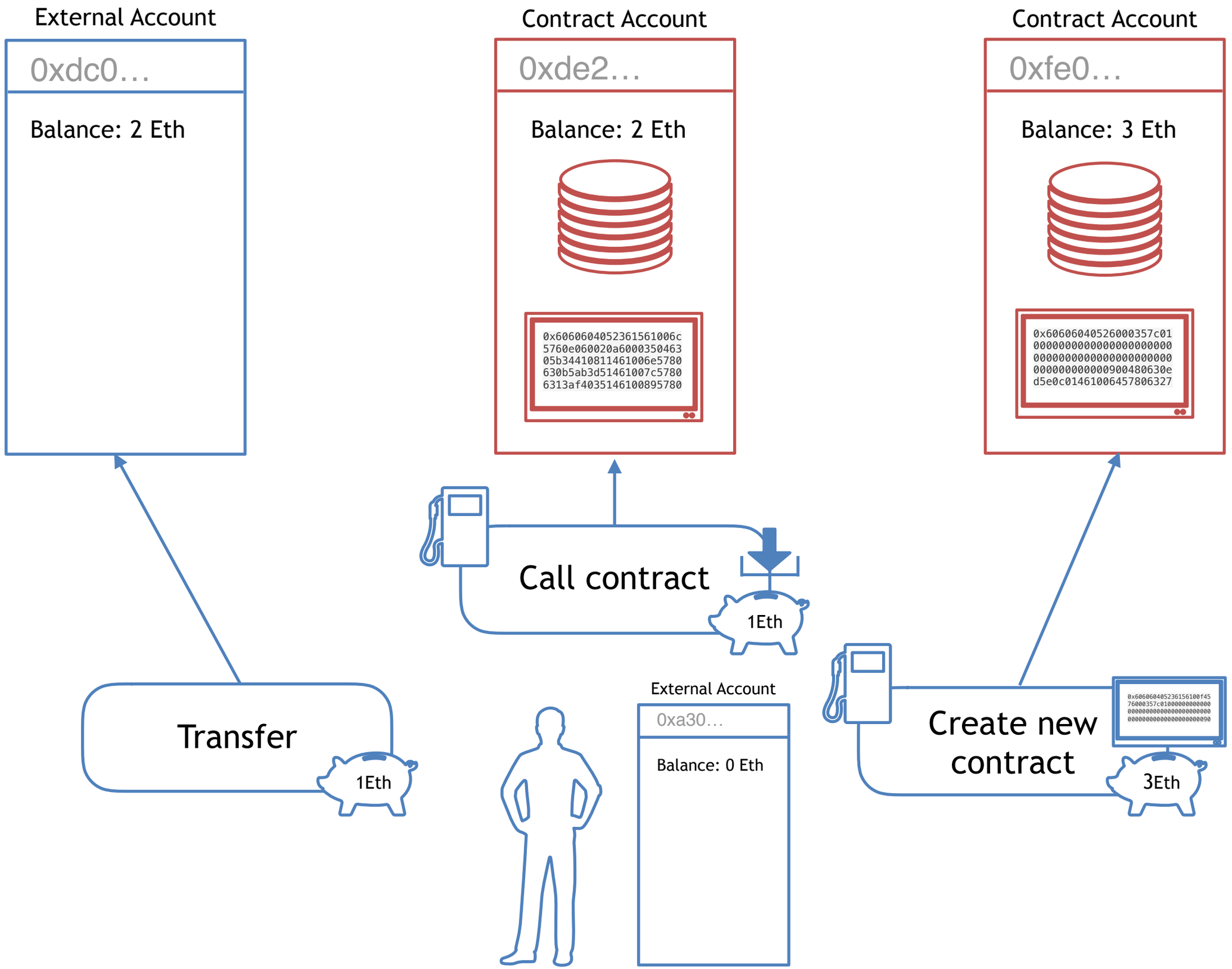
Balance: 1 Eth

Create new contract

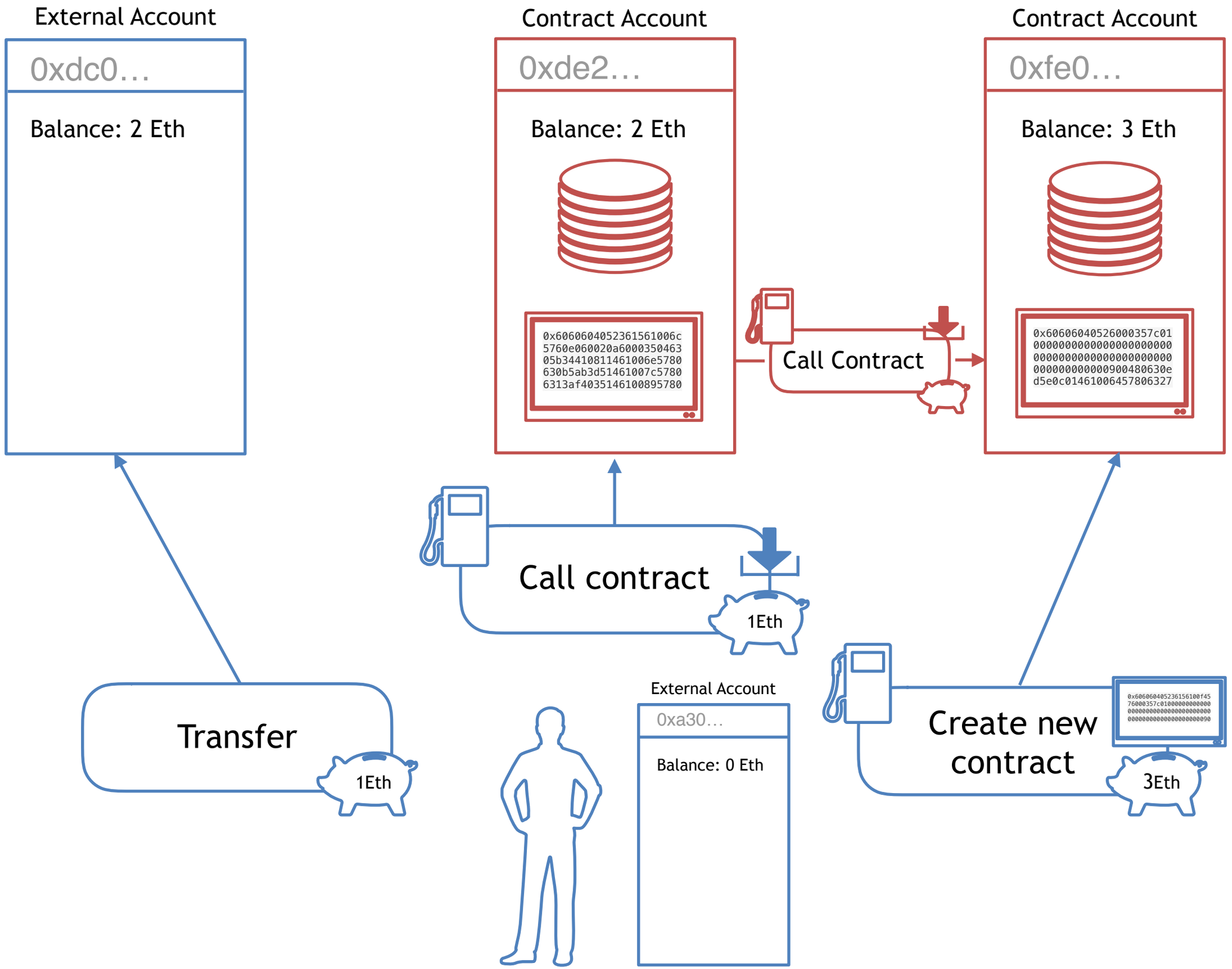


3Eth

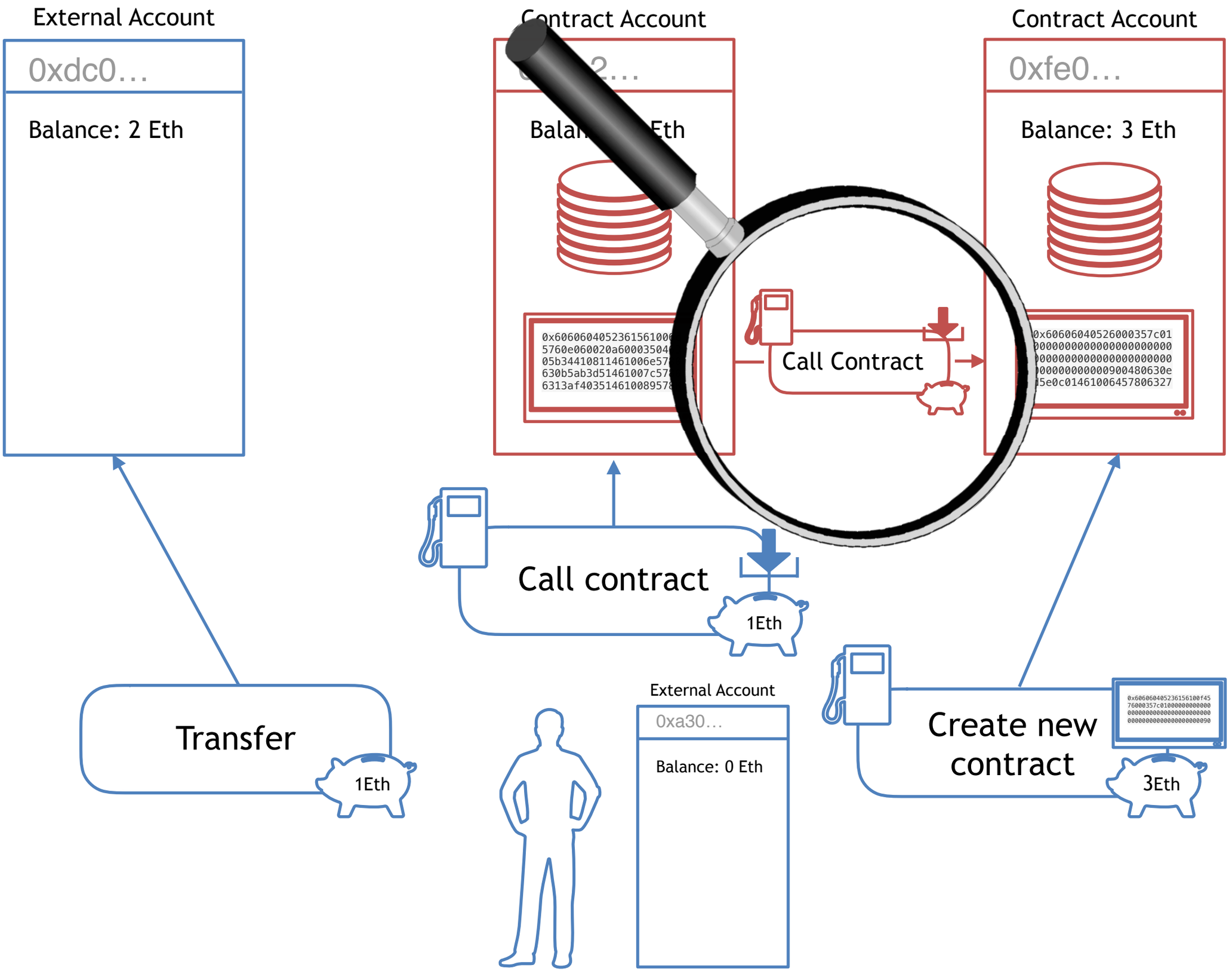
Overview on Ethereum



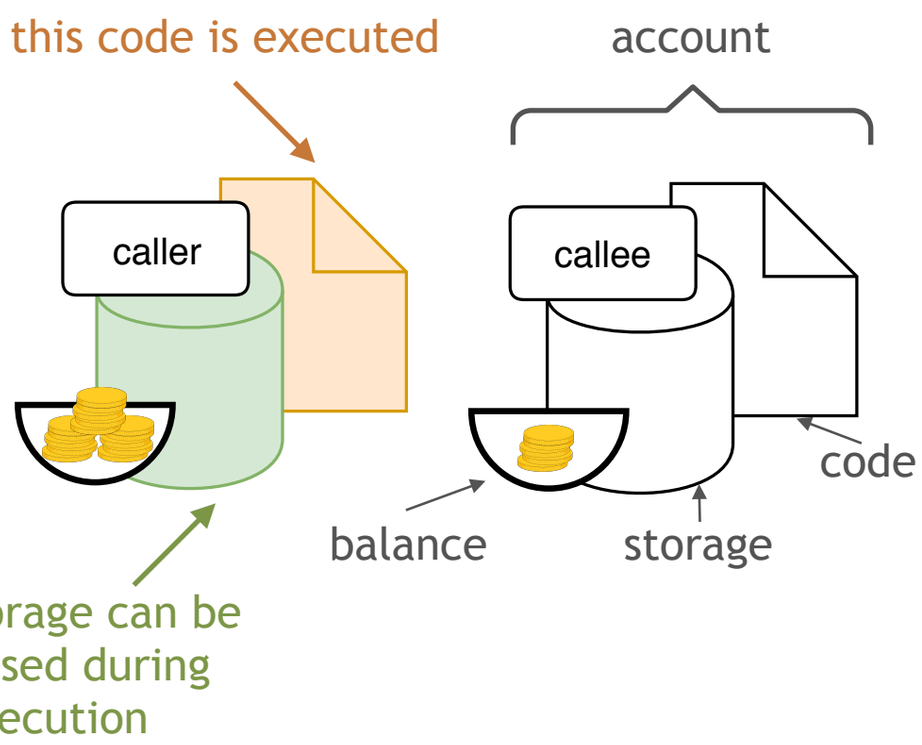
Overview on Ethereum



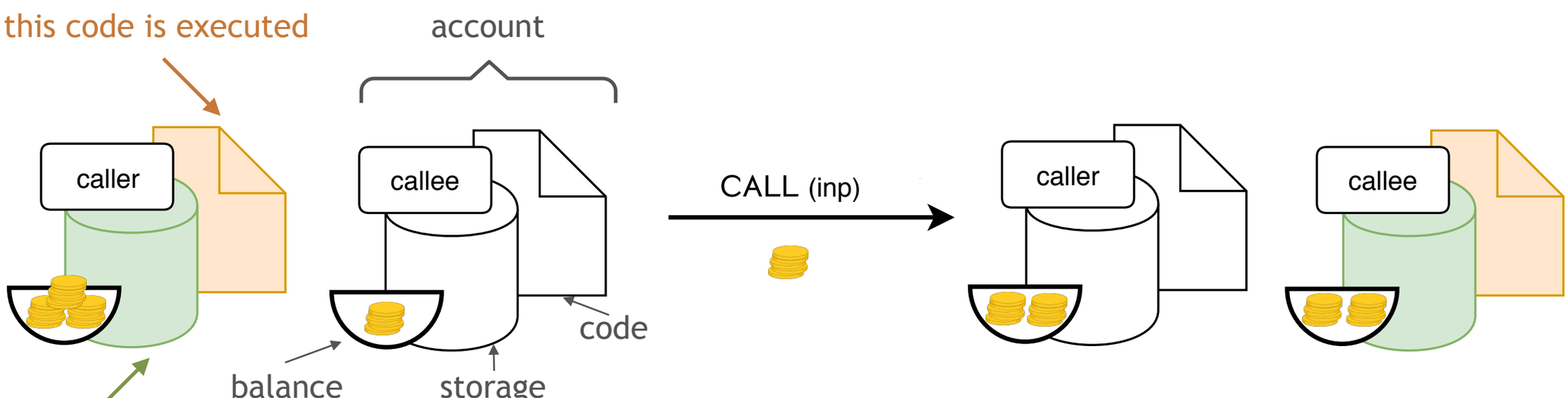
Overview on Ethereum



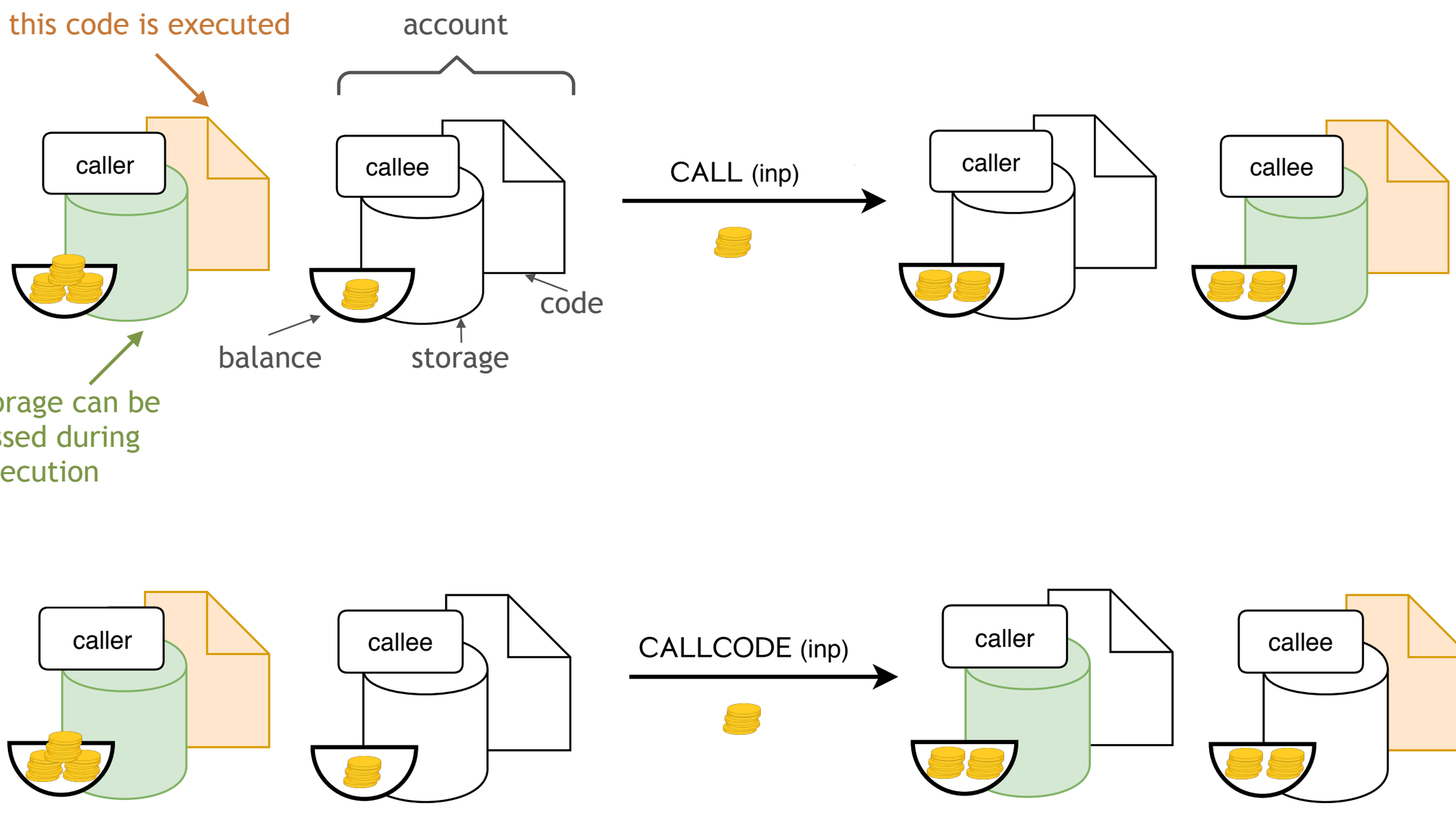
Call flavours



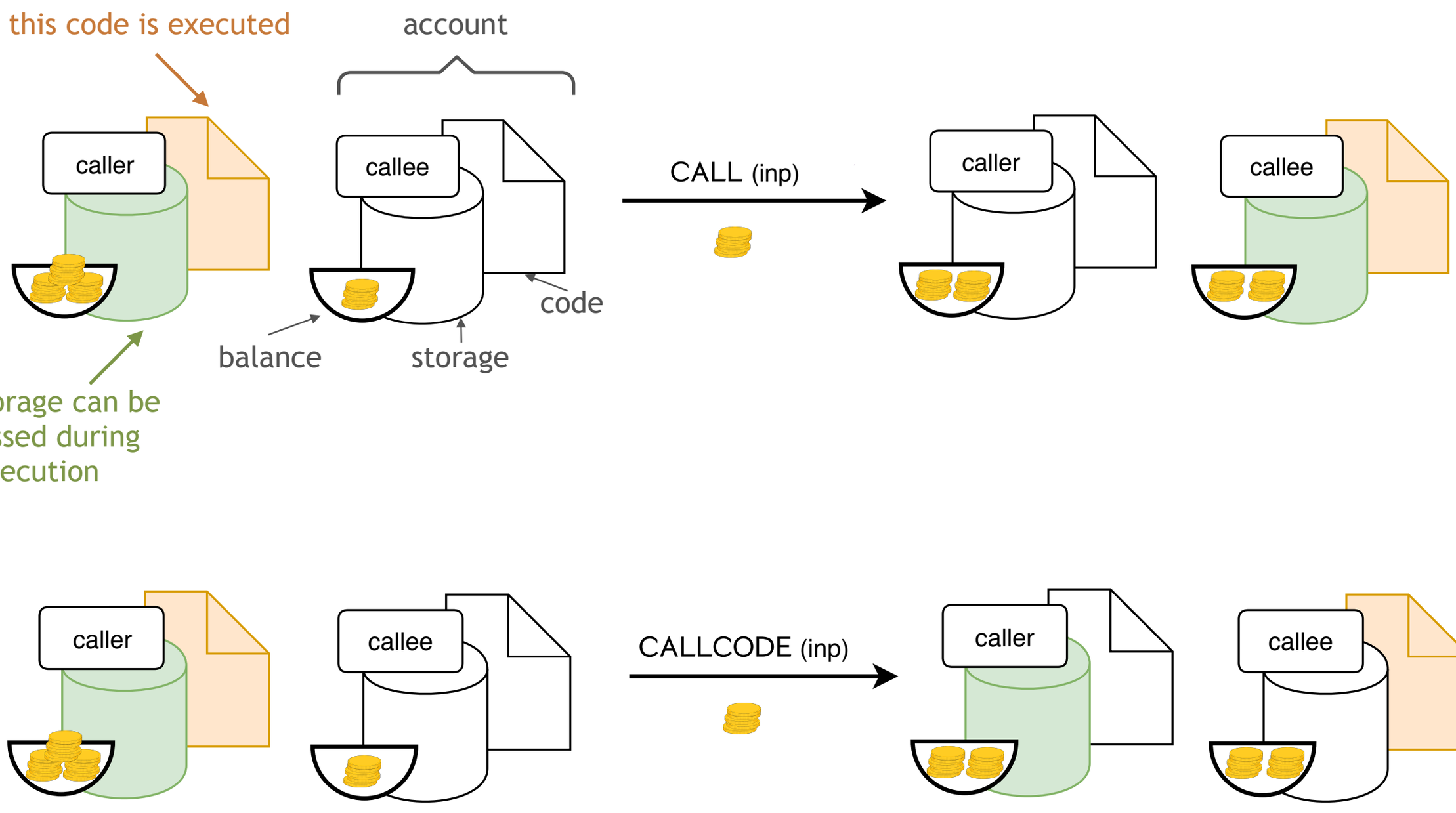
Call flavours



Call flavours



Call flavours



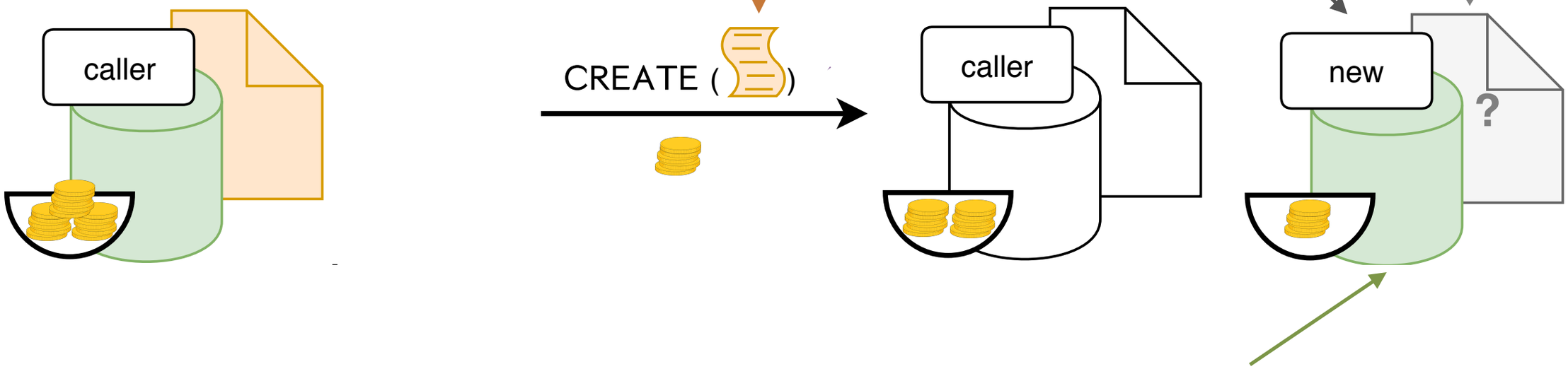
Callee's code can modify the state of the caller!

Contract creation

Upon successful execution, the initialisation code returns as output a code that will be (from that point on) attached to the new account

Initialisation code is given as argument

new account without code is created



while executing initialisation code the storage of the newly created account can be accessed (e.g. global variables can be initialised)

Outline

Introduction to Ethereum

Semantics of EVM bytecode

Static Analysis of EVM bytecode

We go for a slightly simplified setting

(Only plain calls, simplified gas treatment, etc.)

Full treatment in…

A Semantic Framework for the Security Analysis of Ethereum smart contracts

Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind

TU Wien

{ilya.grishchenko,matteo.maffei,clara.schneidewi

Best paper award at ETAPS'18

EVM Semantics Formalization

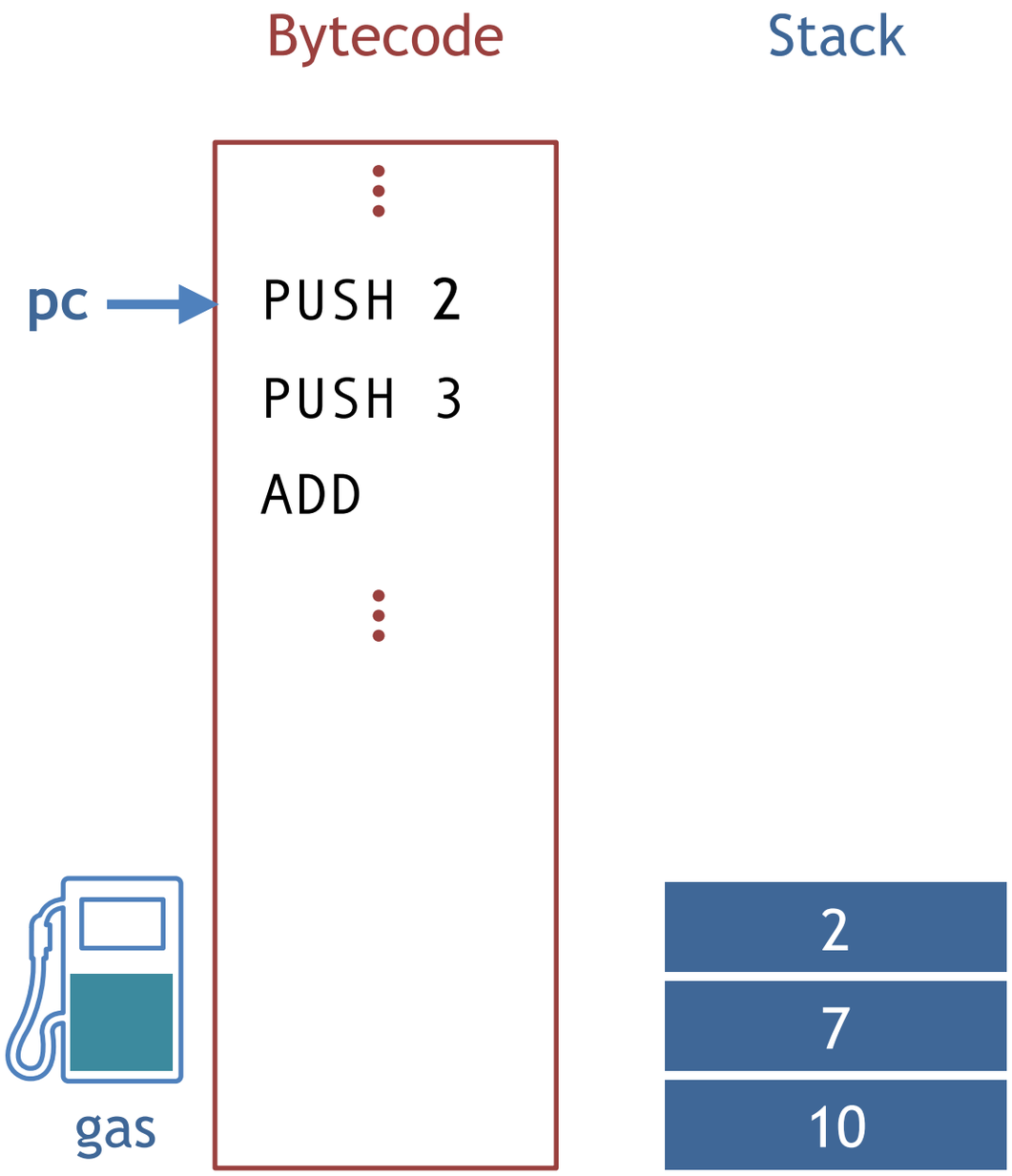
- First complete formalization of EVM bytecode semantics, in the F* proof assistant

- Executable semantics by compilation into OCAML

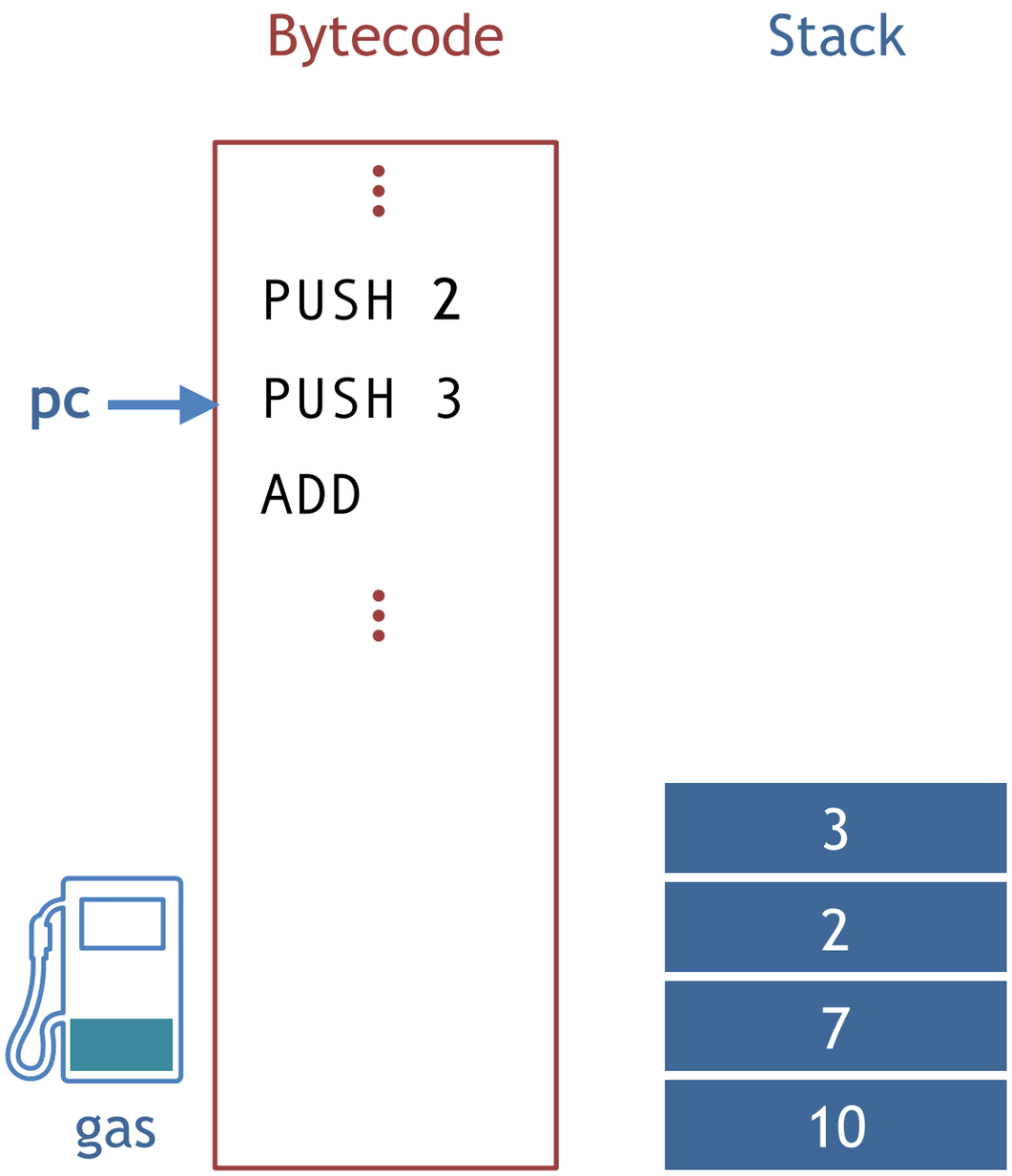
- Tested against the official Ethereum test suite

- While formalizing, we spotted various bugs and imprecisions in previous (in)formal descriptions, including those used in state-of-the-art static analysers (e.g., Oyente)

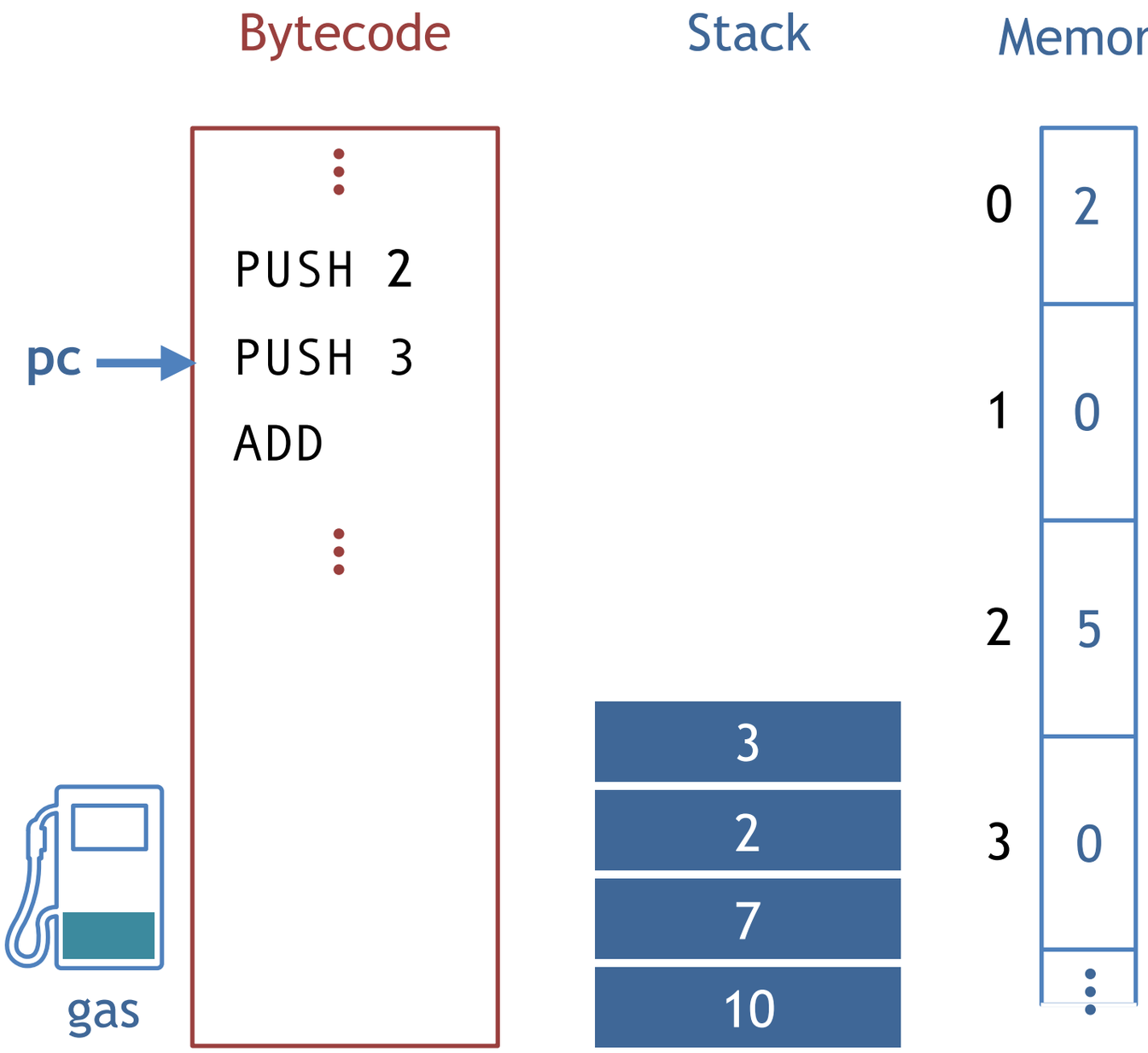
EVM - Layout



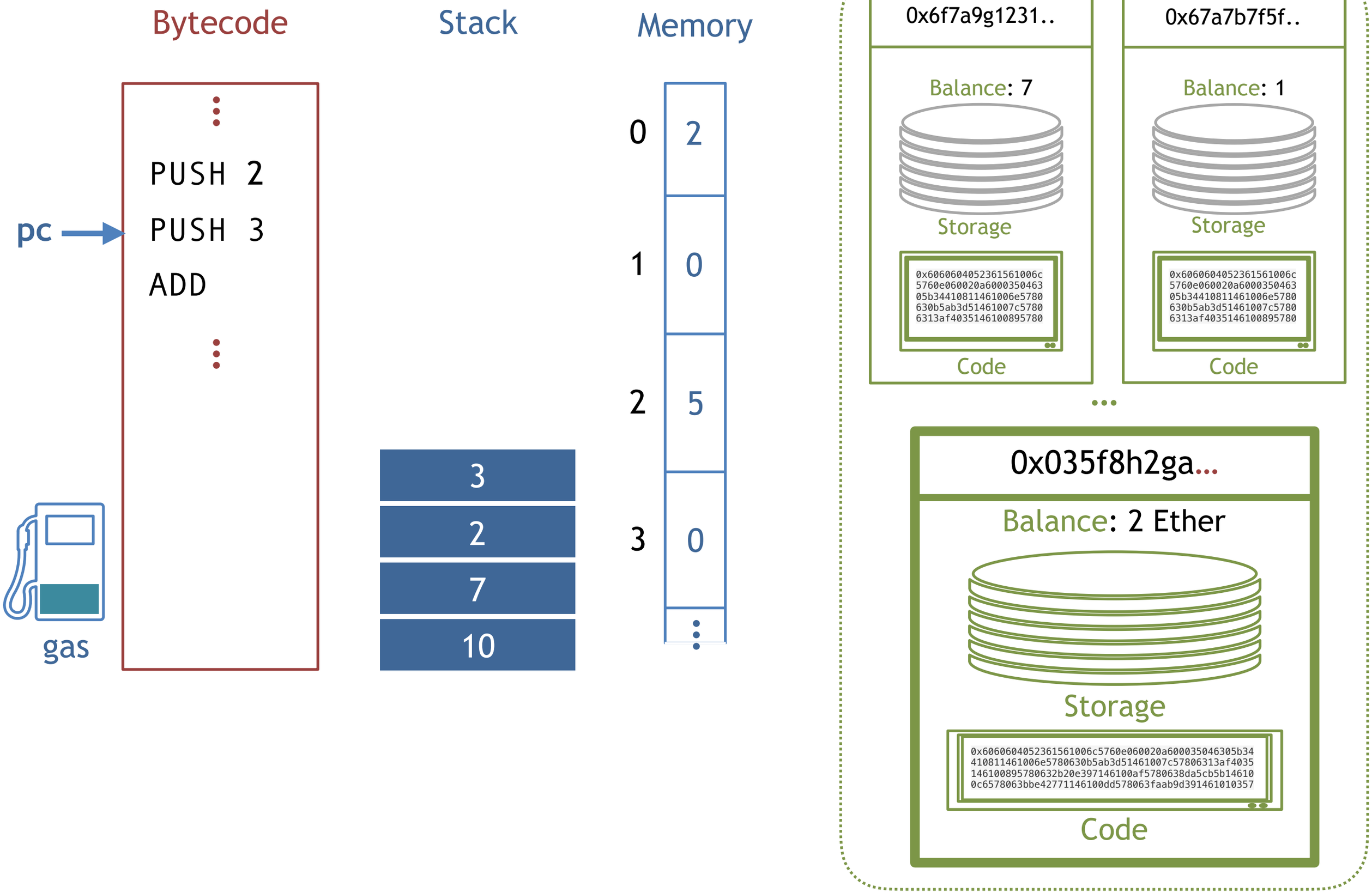
EVM - Layout



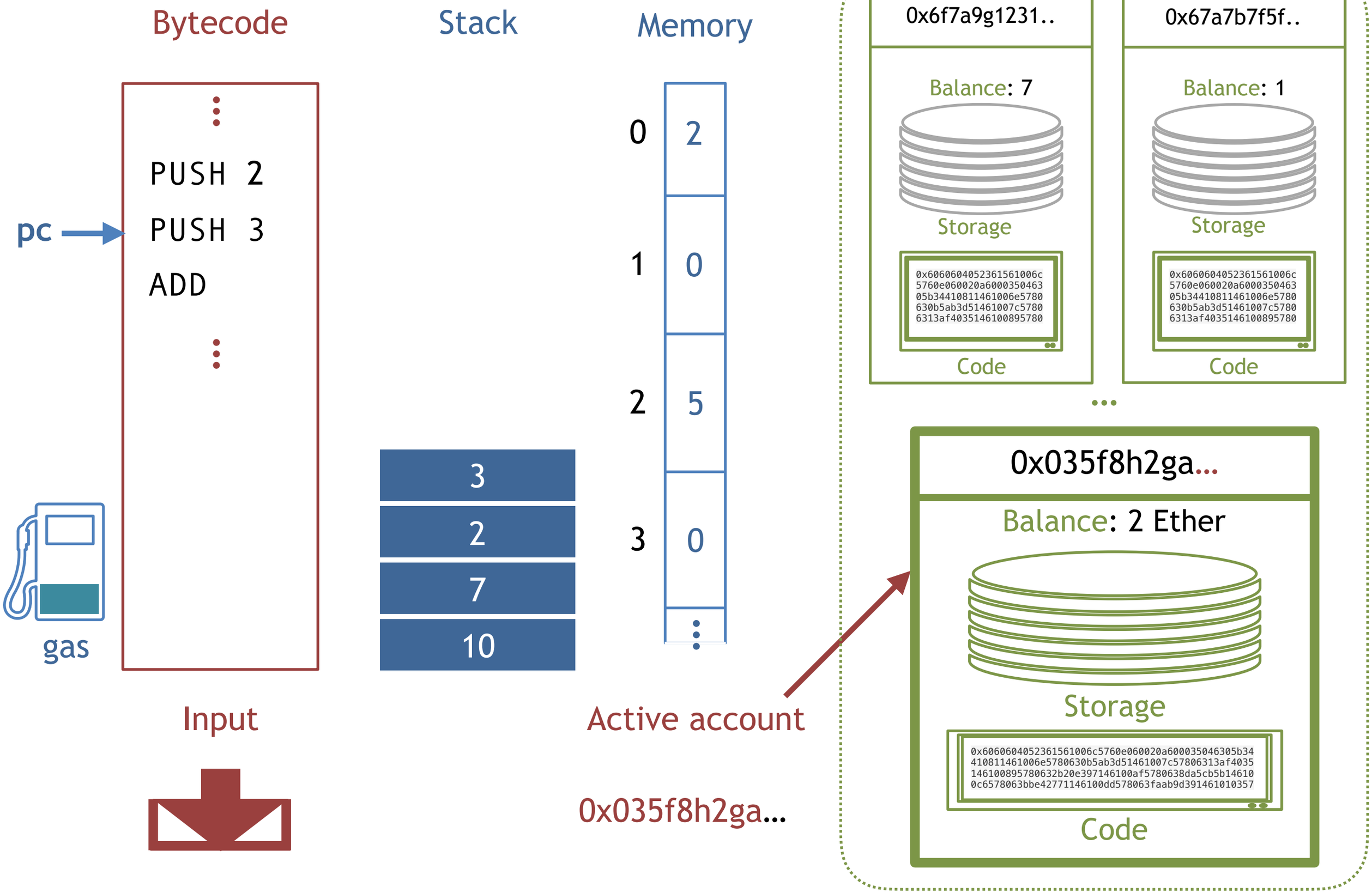
EVM - Layout


Bytecode
Stack
Memory
PUSH 2
PUSH 3
ADD
pc
gas
3
2
7
10
0
2
1
0
2
5
3
0

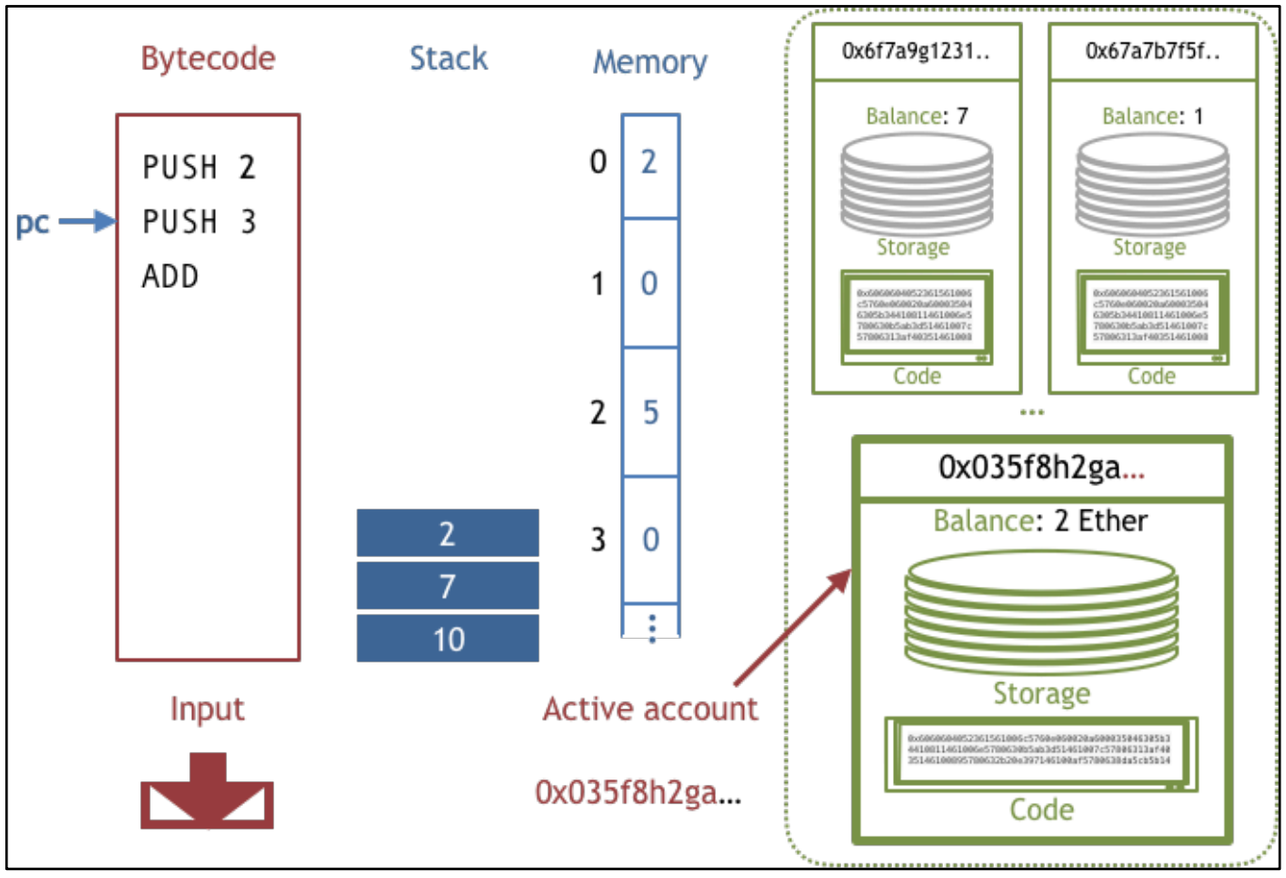
EVM - Layout


Bytecode
Stack
Memory
PUSH 2
PUSH 3
ADD
pc
gas
3
2
7
10
0
1
2
3
2
0
5
0
0x6060604052361561006c5760e060020a6000350463
05b34410811461006e5780630b5ab3d51461007c5780
6313af4035146100895780
Code
Balance: 7
Storage
0x6060604052361561006c5760e060020a6000350463
05b34410811461006e5780630b5ab3d51461007c5780
6313af4035146100895780
Code
Balance: 1
Storage
0x035f8h2ga...
Balance: 2 Ether
Storage
0x6060604052361561006c5760e060020a600035046305b34
410811461006e5780630b5ab3d51461007c57806313af4035
14610089578632b20e397146100af5780638da5cb5b14610
0c6578063bbe42771146100dd578063faab9d391461010357
Code

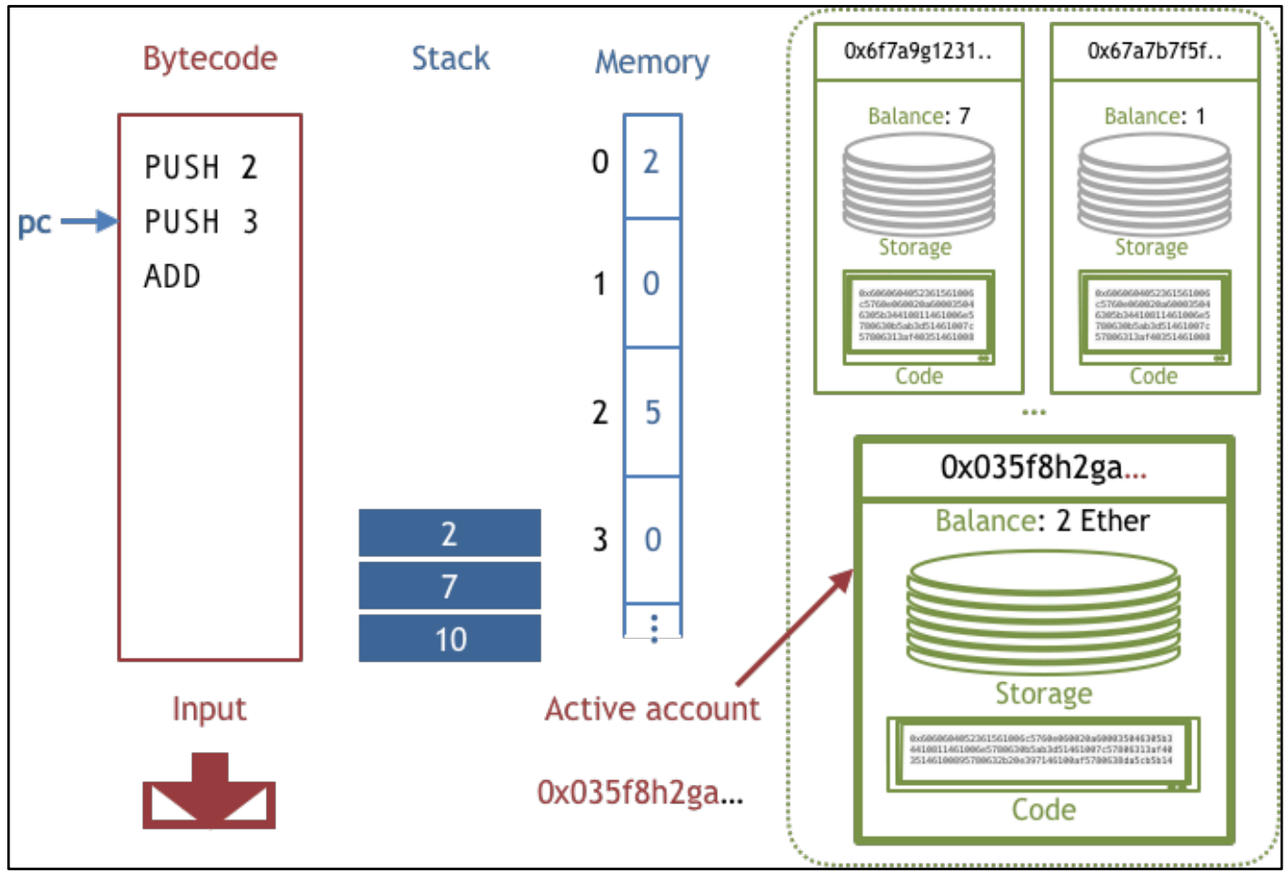
EVM - Layout


Bytecode
PUSH 2
PUSH 3
ADD
pc
gas
Input
Stack
3
2
7
10
Memory
0
2
1
0
2
5
3
0
Active account
0x035f8h2ga...
0x6f7a9g1231..
Balance: 7
Storage
0x60606040523615610 06c5760e060020a6000350463 05b34410811461006e5780 630b5ab3d51461007c5780 6313af4035146100895780
Code
0x67a7b7f5f..
Balance: 1
Storage
0x60606040523615610 06c5760e060020a6000350463 05b34410811461006e5780 630b5ab3d51461007c5780 6313af4035146100895780
Code
0x035f8h2ga...
Balance: 2 Ether
Storage
0x6060604052361561006c5760e060020a600035046305b34 410811461006e5780630b5ab3d51461007c57806313af4035 14610089578632b20e397146100af5780638da5cb5b14610 0c6578063bbe42771146100dd578063faab9d391461010357
Code

Execution states



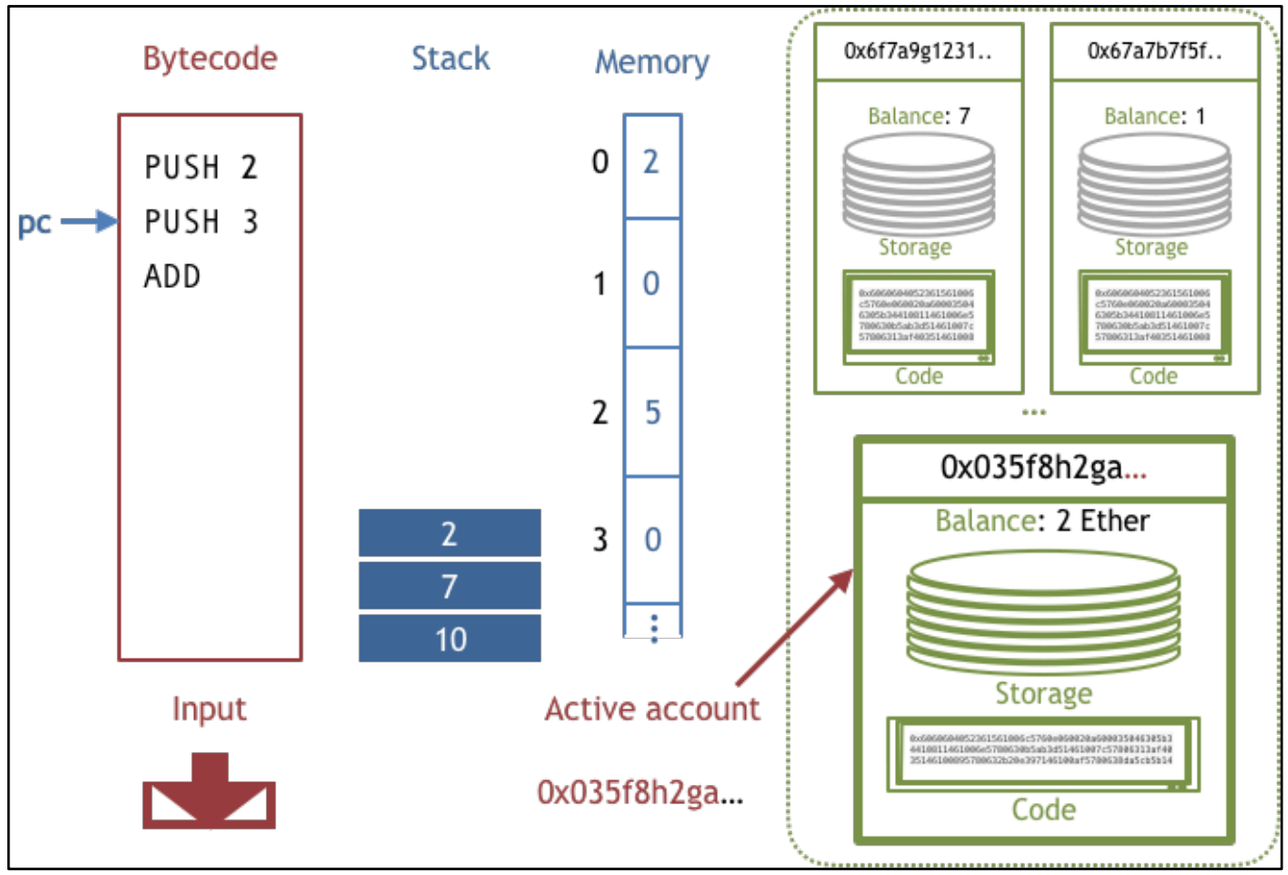
Execution states



$$(\mu, \iota, \sigma)$$

machine state — **execution environment** — **global state**

Execution states



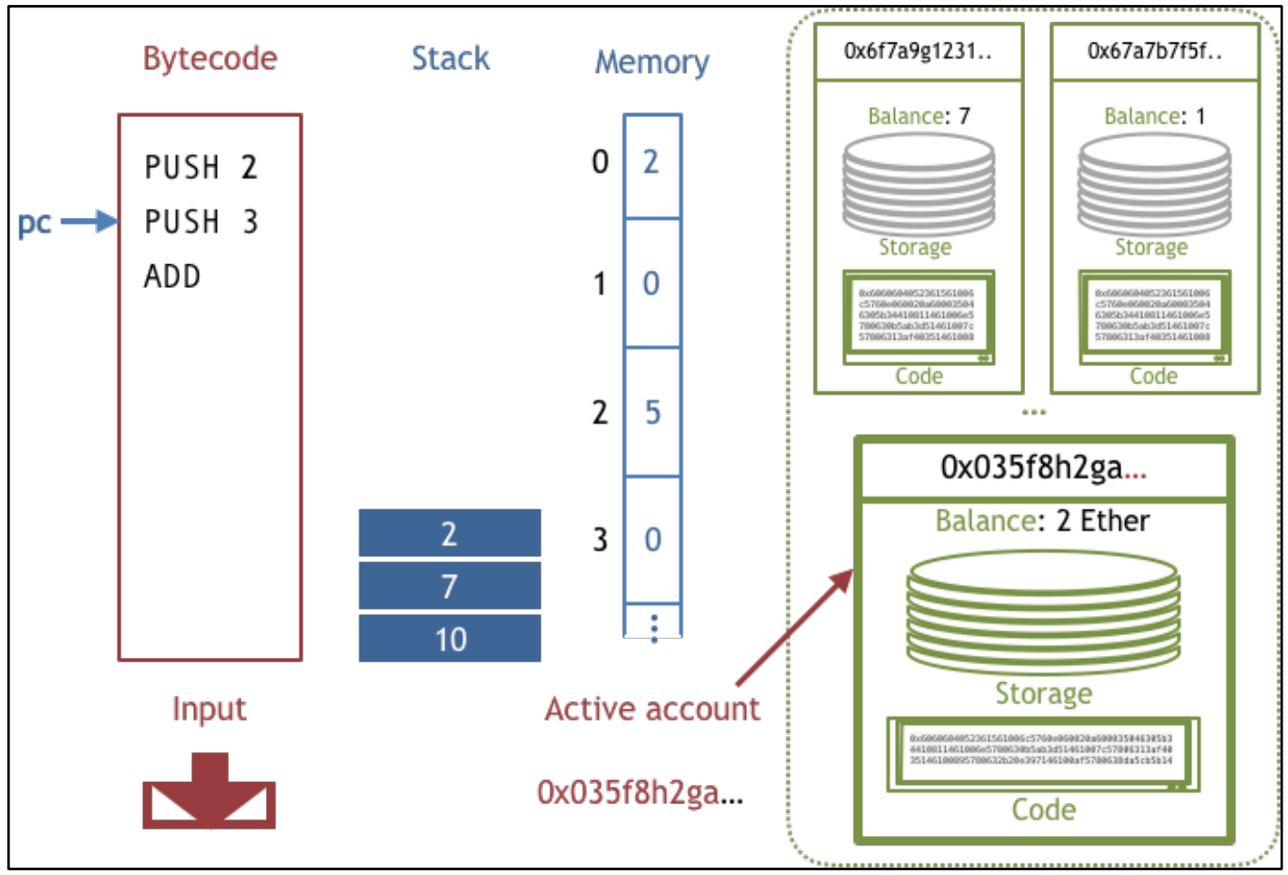
$$(\mu, \iota, \sigma)$$

machine state — **execution environment** — **global state**

$$(gas, pc, m, s)$$

- gas: remaining gas
- pc: program counter
- m: memory
- s: stack

Execution states



$$(\mu, \iota, \sigma)$$

machine state

$$(gas, pc, m, s)$$

- remaining gas
- memory
- program counter
- stack

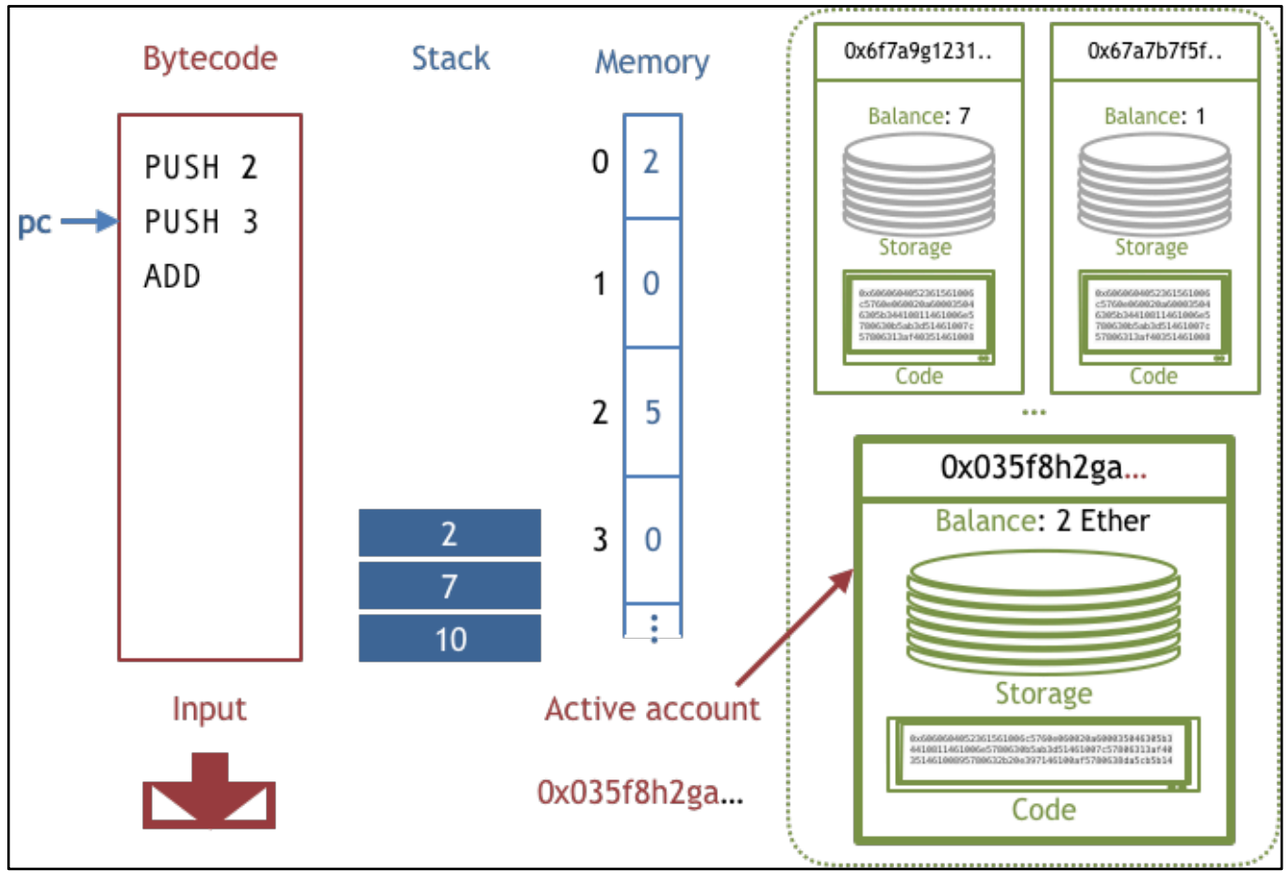
execution environment

$$(actor, input, code)$$

- active account's address
- input to call
- executed code

global state

Execution states



$$(\mu, \iota, \sigma)$$

machine state

$$(gas, pc, m, s)$$

- remaining gas
- program counter
- memory
- stack

execution environment

$$(actor, input, code)$$

- active account's address
- input to call
- executed code

global state

$$\sigma(a) = (b, stor, code)$$

- account address
- balance
- storage
- account code

Small-step semantics

$$
\begin{array}{rlcll}
\text{Call stacks} & \mathbb{S} & \ni & S & := & EXC :: S_{plain} \mid HALT(\sigma, d, g) :: S_{plain} \mid S_{plain} \\
\text{Plain call stacks} & \mathbb{S}_{plain} & \ni & S_{plain} & := & (\mu, \iota, \sigma) :: S_{plain} \mid \epsilon \\
\text{Transaction environments} & \mathcal{T}_{env} & \ni & \Gamma & := & T
\end{array}
$$

Small-step semantics

exceptional halting state
(might be entered e.g. as execution ran out of gas)

$$
\begin{array}{rlcll}
\text{Call stacks} & \mathbb{S} & \ni & S & := \boxed{EXC} :: S_{plain} \mid HALT(\sigma, d, g) :: S_{plain} \mid S_{plain} \\
\text{Plain call stacks} & \mathbb{S}_{plain} & \ni & S_{plain} & := (\mu, \iota, \sigma) :: S_{plain} \mid \epsilon \\
\text{Transaction environments} & \mathcal{T}_{env} & \ni & \Gamma & := T
\end{array}
$$

Small-step semantics

exceptional halting state
(might be entered e.g. as execution ran out of gas)

regular halting state
(holds resulting global state **σ**, return value **d** and remaining gas **g**)

$$
\begin{array}{rlcll}
\text{Call stacks} & \mathbb{S} & \ni & S & := \boxed{EXC} :: S_{plain} \mid \boxed{HALT(\sigma, d, g)} :: S_{plain} \mid S_{plain} \\
\text{Plain call stacks} & \mathbb{S}_{plain} & \ni & S_{plain} & := (\mu, \iota, \sigma) :: S_{plain} \mid \epsilon \\
\text{Transaction environments} & \mathcal{T}_{env} & \ni & \Gamma & := T
\end{array}
$$

Small-step semantics

exceptional halting state
(might be entered e.g. as execution ran out of gas)

regular halting state
(holds resulting global state **σ**, return value **d** and remaining gas **g**)

$$
\begin{array}{rrcll}
\text{Call stacks} & \mathbb{S} & \ni & S & := \boxed{EXC} :: S_{plain} \mid \boxed{HALT(\sigma, d, g)} :: S_{plain} \mid S_{plain} \\
\text{Plain call stacks} & \mathbb{S}_{plain} & \ni & S_{plain} & := (\mu, \iota, \sigma) :: S_{plain} \mid \epsilon \\
\text{Transaction environments} & \mathcal{T}_{env} & \ni & \Gamma & := \boxed{T}
\end{array}
$$

block timestamp

Small-step semantics

exceptional halting state
(might be entered e.g. as execution ran out of gas)

regular halting state
(holds resulting global state **σ**, return value **d** and remaining gas **g**)

$$
\begin{array}{rrcll}
\text{Call stacks} & \mathbb{S} & \ni & S & := \boxed{EXC} :: S_{plain} \mid \boxed{HALT(\sigma, d, g)} :: S_{plain} \mid S_{plain} \\
\text{Plain call stacks} & \mathbb{S}_{plain} & \ni & S_{plain} & := (\mu, \iota, \sigma) :: S_{plain} \mid \epsilon \\
\text{Transaction environments} & \mathcal{T}_{env} & \ni & \Gamma & := \boxed{T}
\end{array}
$$

block timestamp

Small step relation

$$\Gamma \models S \rightarrow S'$$

describes how a call stack S evolves within one step of execution under transaction environment $\mathbf{\Gamma}$

Simplified EVM bytecode

Simplified EVM bytecode

- Arithmetic, Logical & Comparison instructions:
  `ADD, MUL, LEQ, NOT, AND, OR`

Simplified EVM bytecode

- Arithmetic, Logical & Comparison instructions:
  `ADD, MUL, LEQ, NOT, AND, OR`
- Control flow instructions:
  `JUMP` *`pc`*`, JUMPI` *`pc`*

Simplified EVM bytecode

- Arithmetic, Logical & Comparison instructions:
  `ADD, MUL, LEQ, NOT, AND, OR`
- Control flow instructions:
  `JUMP` *`pc`*`, JUMPI` *`pc`*
- Stack modifying instructions:
  `PUSH` *`x`*`, POP`

Simplified EVM bytecode

- Arithmetic, Logical & Comparison instructions:
  `ADD, MUL, LEQ, NOT, AND, OR`
- Control flow instructions:
  `JUMP` *`pc`*`, JUMPI` *`pc`*
- Stack modifying instructions:
  `PUSH` *`x`*`, POP`
- (Local) Memory instructions
  `MSTORE, MLOAD`

Simplified EVM bytecode

- Arithmetic, Logical & Comparison instructions:
  `ADD, MUL, LEQ, NOT, AND, OR`

- Control flow instructions:
  `JUMP` *`pc`*`, JUMPI` *`pc`*

- Stack modifying instructions:
  `PUSH` *`x`*`, POP`

- (Local) Memory instructions
  `MSTORE, MLOAD`

- (Global) Storage instructions
  `SSTORE, SLOAD`

Simplified EVM bytecode

- Arithmetic, Logical & Comparison instructions:
  `ADD, MUL, LEQ, NOT, AND, OR`

- Control flow instructions:
  `JUMP` *`pc`*`, JUMPI` *`pc`*

- Stack modifying instructions:
  `PUSH` *`x`*`, POP`

- (Local) Memory instructions
  `MSTORE, MLOAD`

- (Global) Storage instructions
  `SSTORE, SLOAD`

- Environment access
  (global state + execution environment + transaction environment)
  `BALANCE, TIMESTAMP, INPUT, ADDRESS`

Simplified EVM bytecode

- Arithmetic, Logical & Comparison instructions:
  `ADD, MUL, LEQ, NOT, AND, OR`

- Control flow instructions:
  `JUMP` *`pc`*`, JUMPI` *`pc`*

- Stack modifying instructions:
  `PUSH` *`x`*`, POP`

- (Local) Memory instructions
  `MSTORE, MLOAD`

- (Global) Storage instructions
  `SSTORE, SLOAD`

- Environment access
  (global state + execution environment + transaction environment)
  `BALANCE, TIMESTAMP, INPUT, ADDRESS`

- Contract Calls:
  `CALL`

Simplified EVM bytecode

- Arithmetic, Logical & Comparison instructions:
  `ADD, MUL, LEQ, NOT, AND, OR`

- Control flow instructions:
  `JUMP` *`pc`*`, JUMPI` *`pc`*

- Stack modifying instructions:
  `PUSH` *`x`*`, POP`

- (Local) Memory instructions
  `MSTORE, MLOAD`

- (Global) Storage instructions
  `SSTORE, SLOAD`

- Environment access
  (global state + execution environment + transaction environment)
  `BALANCE, TIMESTAMP, INPUT, ADDRESS`

- Contract Calls:
  `CALL`

- Halting:
  `RETURN, STOP`

Simple stack operations

The instruction at μ.pc of code ι.code is ADD

$$\frac{\mu.\text{gas} >= 1 \qquad \mu.\text{s} = a :: b :: s \qquad \omega_{\mu,\iota} = \text{ADD} \qquad \mu' = \mu[\text{s} \rightarrow (a+b) :: s][\text{pc} \mathrel{+}= 1][\text{gas} \mathrel{-}= 1]}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow (\mu', \iota, \sigma) :: S}$$

preconditions are checked: enough gas available + enough element on the stack

machine state is updated

Simple stack operations

The instruction at μ.pc of code ι.code is ADD

$$\frac{\mu.\mathsf{gas} >= 1 \qquad \mu.\mathsf{s} = a :: b :: s \qquad \omega_{\mu,\iota} = \mathsf{ADD} \qquad \mu' = \mu[\mathsf{s} \rightarrow (a+b) :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 1]}{\Gamma \vDash (\mu, \iota, \sigma) :: S \ \rightarrow (\mu', \iota, \sigma) :: S}$$

preconditions are checked: enough gas available + enough element on the stack

machine state is updated

$$\frac{\omega_{\mu,\iota} = \mathsf{ADD} \qquad |\mu.\mathsf{s}| < 2}{\Gamma \vDash (\mu, \iota, \sigma) :: S \ \rightarrow EXC :: S}$$

in case of a stack underflow the execution halts exceptionally

Simple stack operations

The instruction at μ.pc of code ι.code is ADD

$$\frac{\mu.\mathsf{gas} >= 1 \qquad \mu.\mathsf{s} = a :: b :: s \qquad \omega_{\mu,\iota} = \mathsf{ADD} \qquad \mu' = \mu[\mathsf{s} \rightarrow (a+b) :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 1]}{\Gamma \vDash (\mu, \iota, \sigma) :: S \ \rightarrow (\mu', \iota, \sigma) :: S}$$

preconditions are checked: enough gas available + enough element on the stack

machine state is updated

$$\frac{\omega_{\mu,\iota} = \mathsf{ADD} \qquad |\mu.\mathsf{s}| < 2}{\Gamma \vDash (\mu, \iota, \sigma) :: S \ \rightarrow EXC :: S}$$

in case of a stack underflow the execution halts exceptionally

$$\frac{\mu.\mathsf{gas} < 1}{\Gamma \vDash (\mu, \iota, \sigma) :: S \ \rightarrow EXC :: S}$$

if the execution runs out of gas, the execution halts exceptionally (holds for all instructions)

Memory Access

value on memory address a is written to the stack

$$\frac{\omega_{\mu,\iota} = \text{MLOAD} \qquad \mu.\text{gas} >= 1 \qquad \mu.\text{s} = a :: s \qquad v = \mu.\text{m}[a] \qquad \mu' = \mu[\text{s} \rightarrow v :: s][\text{pc} \mathrel{+}= 1][\text{gas} \mathrel{-}= 1]}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow (\mu', \iota, \sigma) :: S}$$

Memory Access

value on memory address a is written to the stack

$$\frac{\omega_{\mu,\iota} = \text{MLOAD} \qquad \mu.\text{gas} >= 1 \qquad \mu.\text{s} = a :: s \qquad v = \mu.\text{m}[a] \qquad \mu' = \mu[\text{s} \rightarrow v :: s][\text{pc} \mathrel{+}= 1][\text{gas} \mathrel{-}= 1]}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow (\mu', \iota, \sigma) :: S}$$

value b is written to memory address a

$$\frac{\omega_{\mu,\iota} = \text{MSTORE} \qquad \mu.\text{s} = a :: b :: s \qquad \mu.\text{gas} >= 1 \qquad \mu' = \mu[\text{m} \rightarrow \mu.\text{m}[a \rightarrow b]][\text{s} \rightarrow s][\text{pc} \mathrel{+}= 1][\text{gas} \mathrel{-}= 1]}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow (\mu', \iota, \sigma) :: S}$$

Calling

value to be transferred

input data

address in memory to write return value

recipient address

additional precondition: executing account has enough balance

updated global state: money was transferred from ι.*actor* to *to*

$$\frac{\omega_{\mu,\iota} = \text{CALL} \quad \mu.\text{s} = to :: va :: id :: oa :: s \quad \mu.\text{gas} >= 1 \quad \sigma(\iota.\text{actor}).\text{b} \geq va}{\sigma' = \sigma\langle to \to \sigma(to)[\text{b} \mathrel{+}= va]\rangle\langle \iota.\text{actor} \to \sigma(\iota.\text{actor})[\text{b} \mathrel{-}= va]\rangle \quad \mu' = (\mu.\text{gas} - 1, 0, \lambda x.\, 0, \epsilon) \quad \iota' = \iota[\text{actor} \to to][\text{input} \to id][\text{code} \to \sigma(to).\text{code}]}{\Gamma \vDash (\mu, \iota, \sigma) :: S \to (\mu', \iota', \sigma') :: (\mu, \iota, \sigma) :: S}$$

fresh machine state

execution environment for executing recipient contract is initialised

new execution state (representing the execution of recipient contract) is pushed on callstack

The DAO

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  if (donations[msg.sender] > 0)
  { msg.sender.call.value(donations[msg.sender])();
    donations[msg.sender] = 0;
  }
 }
}
```

mapping keeping track of the donations made by different addresses

function for performing donations

function for withdrawing donations

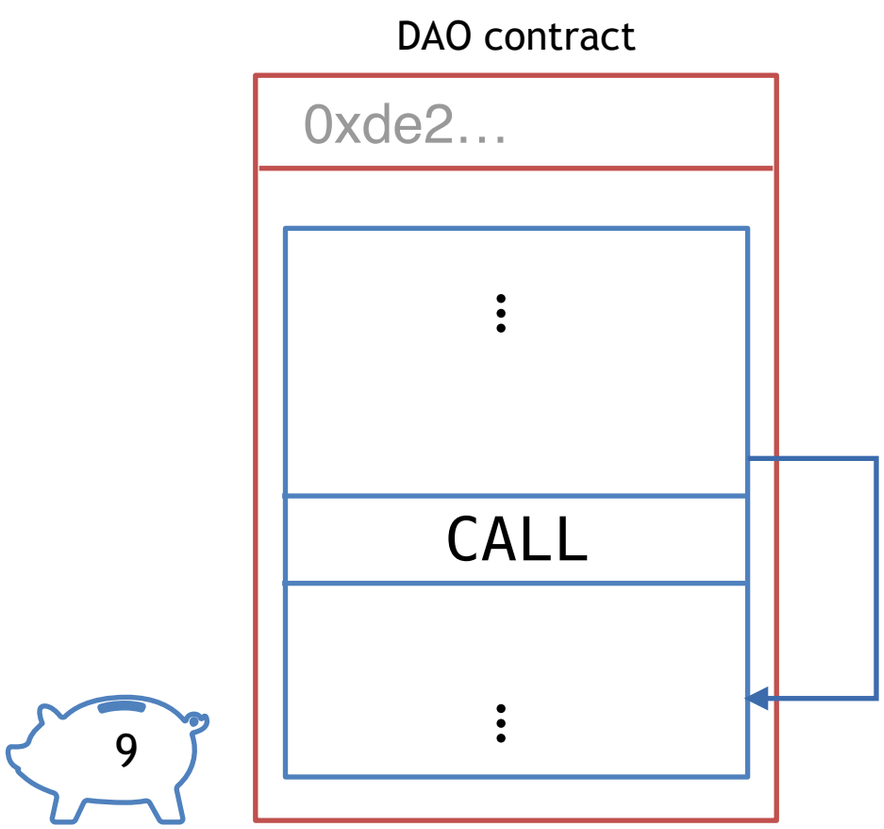
The DAO

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  if (donations[msg.sender] > 0)
  { msg.sender.call.value(donations[msg.sender])();
    donations[msg.sender] = 0;
  }
 }
}
```

mapping keeping track of the donations made by different addresses

function for performing donations

function for withdrawing donations



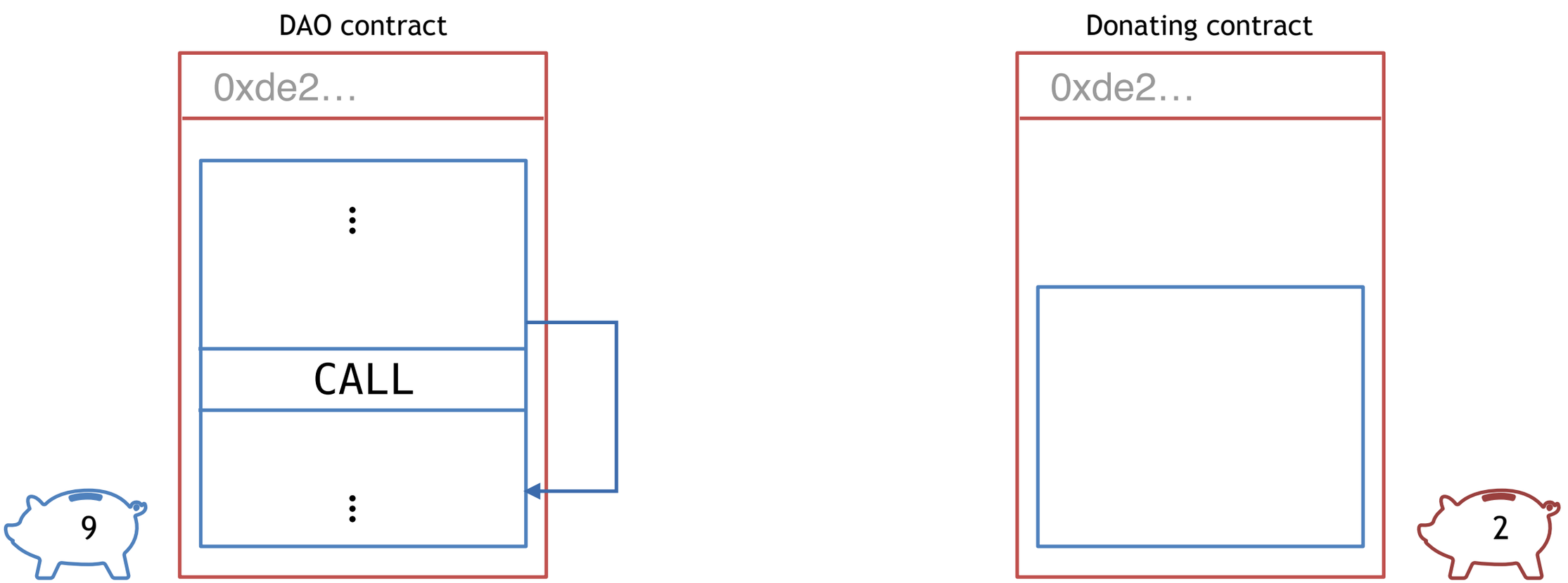
The DAO

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  if (donations[msg.sender] > 0)
  { msg.sender.call.value(donations[msg.sender])();
    donations[msg.sender] = 0;
  }
 }
}
```

mapping keeping track of the donations made by different addresses

function for performing donations

function for withdrawing donations



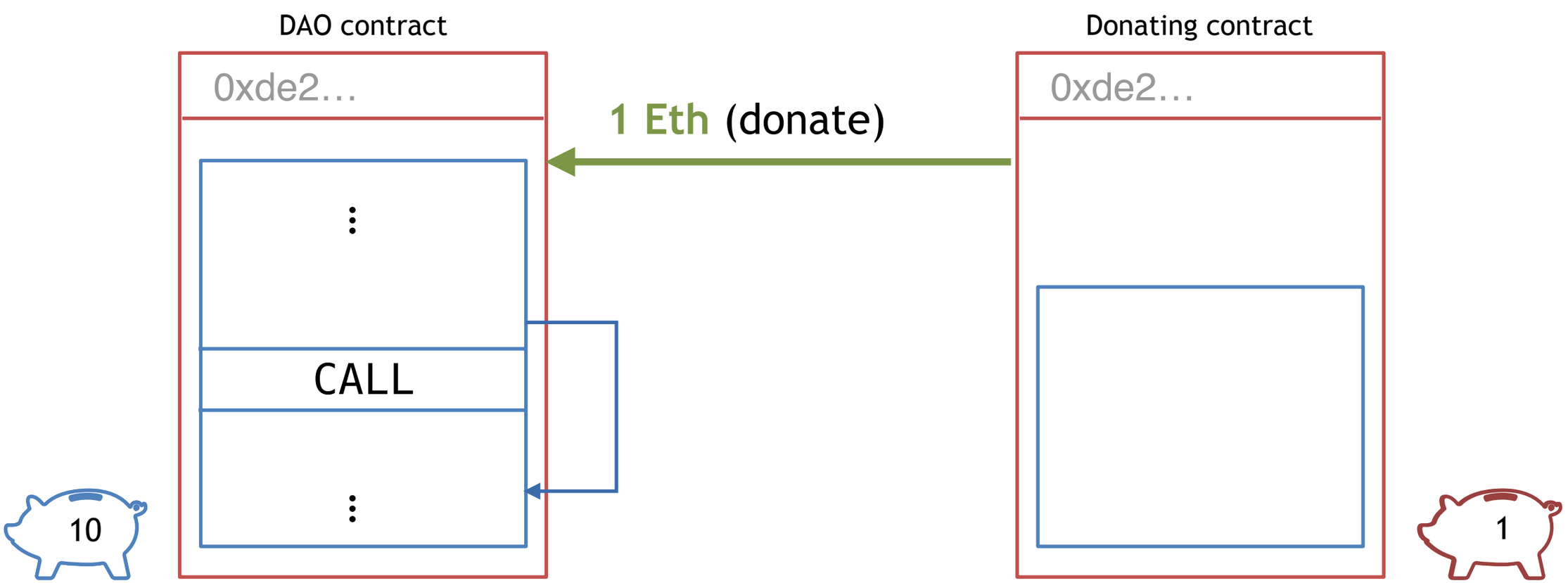
The DAO

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  if (donations[msg.sender] > 0)
  { msg.sender.call.value(donations[msg.sender])();
    donations[msg.sender] = 0;
  }
 }
}
```

mapping keeping track of the donations made by different addresses

function for performing donations

function for withdrawing donations



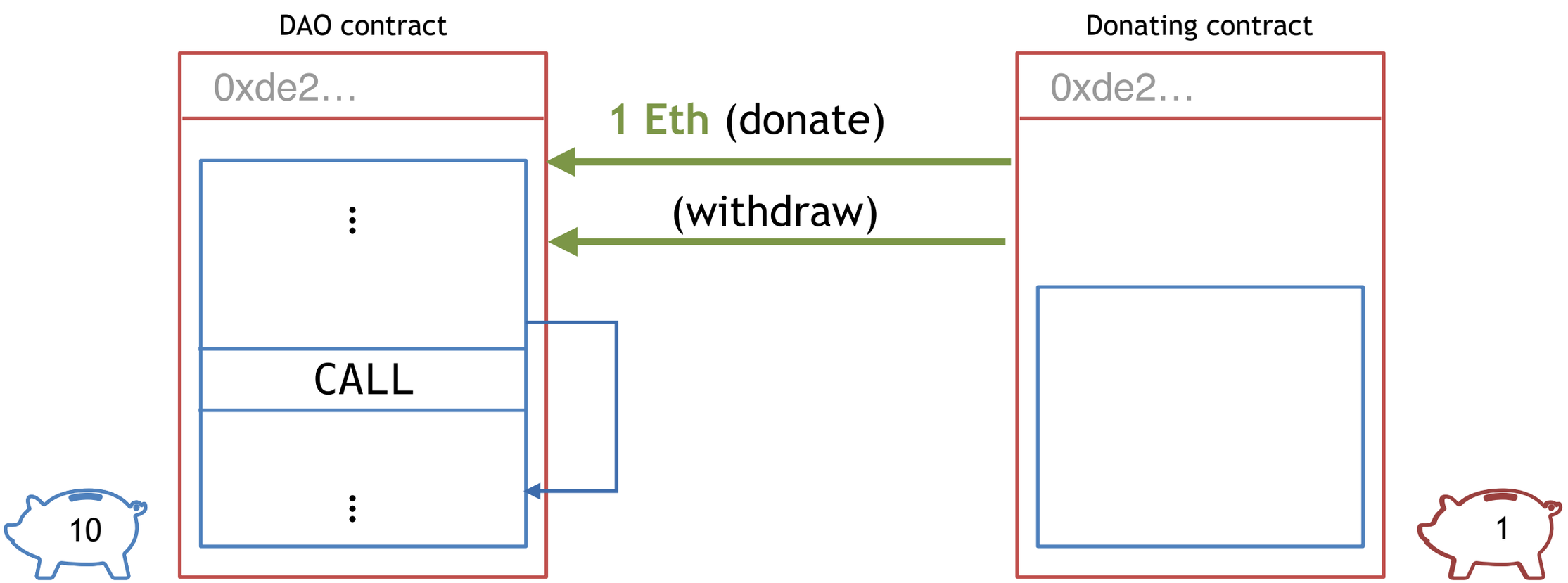
The DAO

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  if (donations[msg.sender] > 0)
  { msg.sender.call.value(donations[msg.sender])();
    donations[msg.sender] = 0;
  }
 }
}
```

mapping keeping track of the donations made by different addresses

function for performing donations

function for withdrawing donations



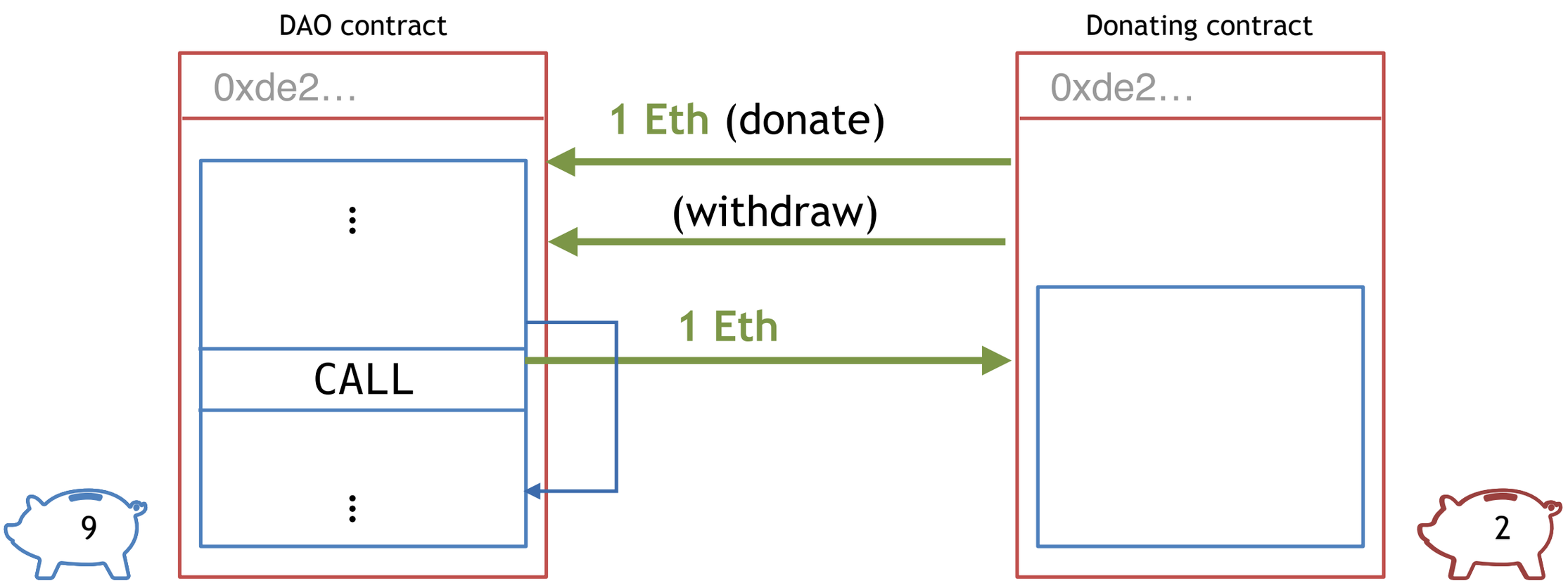
The DAO

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  if (donations[msg.sender] > 0)
  { msg.sender.call.value(donations[msg.sender])();
    donations[msg.sender] = 0;
  }
 }
}
```

mapping keeping track of the donations made by different addresses

function for performing donations

function for withdrawing donations



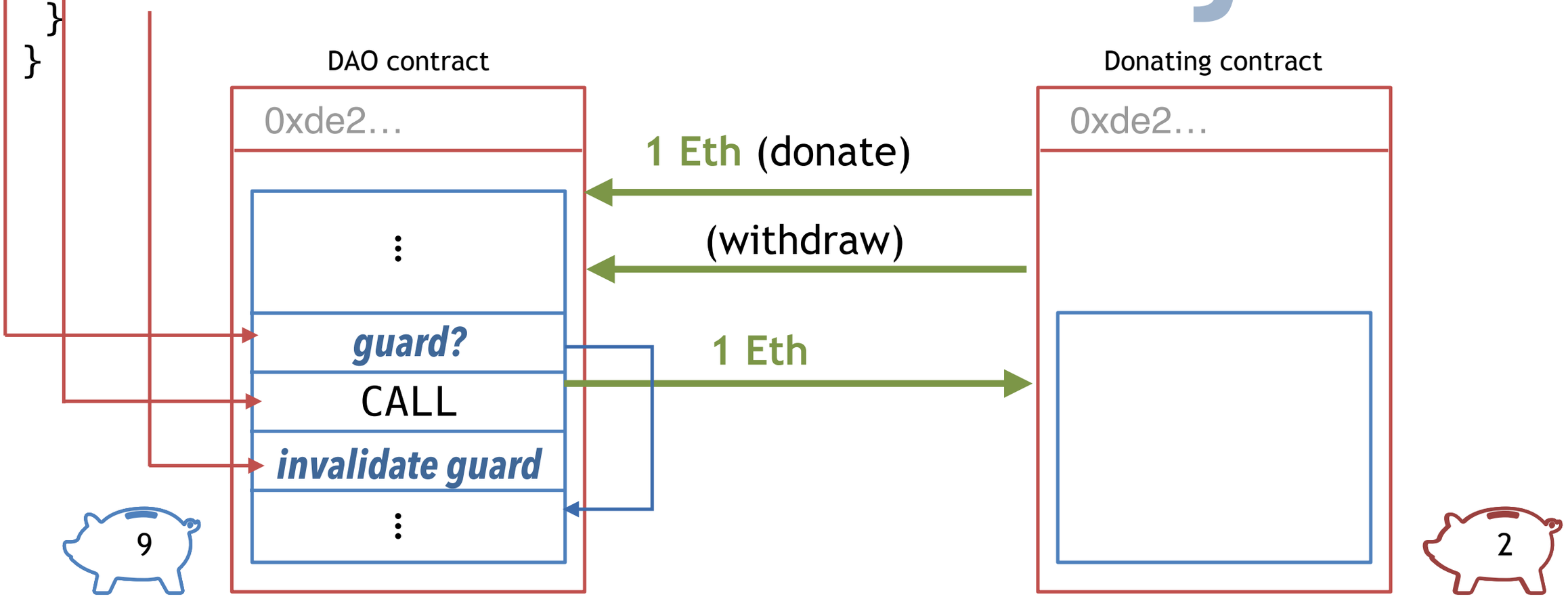
The DAO

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  if (donations[msg.sender] > 0)
  { msg.sender.call.value(donations[msg.sender])();
    donations[msg.sender] = 0;
  }
 }
}
```

mapping keeping track of the donations made by different addresses

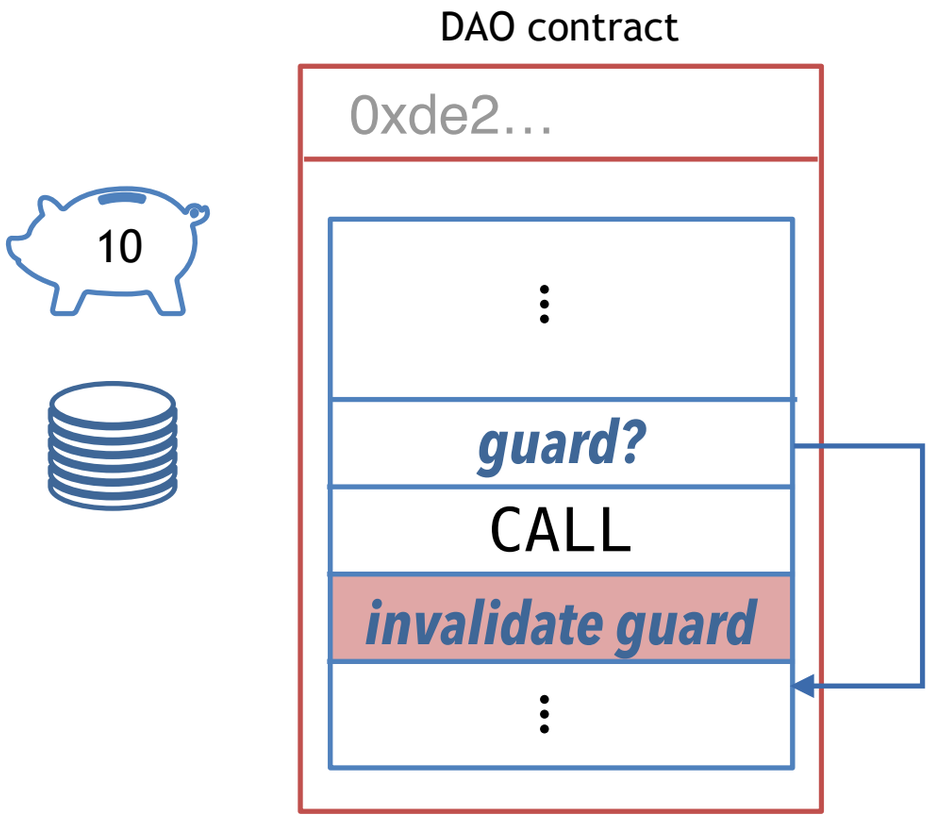
function for performing donations

function for withdrawing donations



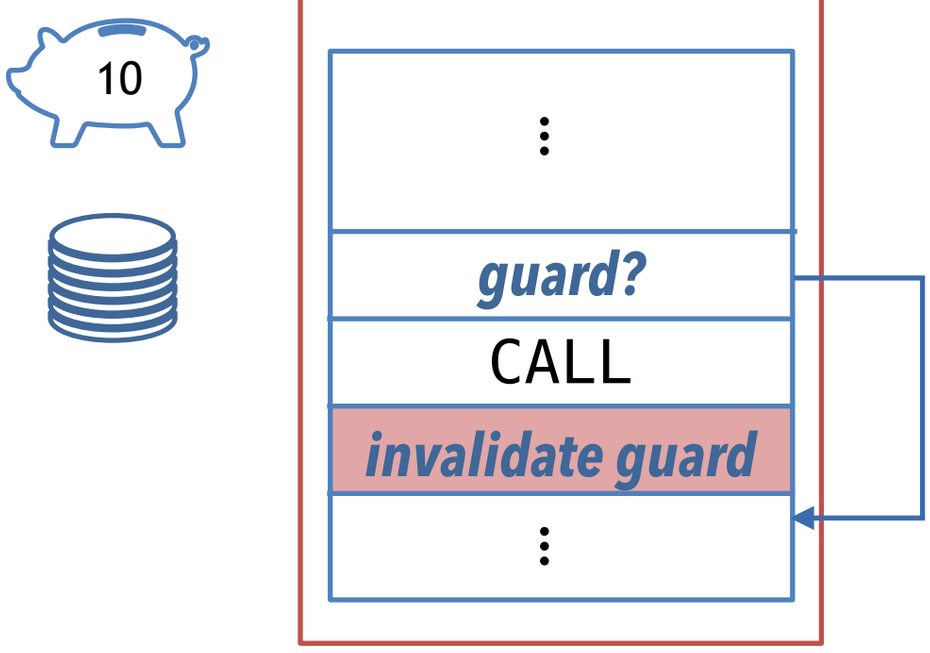
Attack on the DAO

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  if (donations[msg.sender] > 0)
  { msg.sender.call.value(donations[msg.sender])();
    donations[msg.sender] = 0;
  }
 }
```



Attack on the DAO

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  if (donations[msg.sender] > 0)
  { msg.sender.call.value(donations[msg.sender])();
    donations[msg.sender] = 0;
  }
 }
```

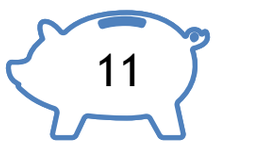


Attack on the DAO

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  if (donations[msg.sender] > 0)
  { msg.sender.call.value(donations[msg.sender])();
    donations[msg.sender] = 0;
  }
 }
}
```

Attack on the DAO

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  if (donations[msg.sender] > 0)
  { msg.sender.call.value(donations[msg.sender])();
    donations[msg.sender] = 0;
  }
 }
```



guard: true

DAO contract

0xde2…

⋮

guard?

CALL

invalidate guard

⋮

1 Eth (donate)

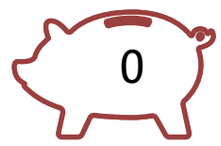
Attacker contract

0xfc2…

⋮

CALL

⋮



Attack on the DAO

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  if (donations[msg.sender] > 0)
  { msg.sender.call.value(donations[msg.sender])();
    donations[msg.sender] = 0;
  }
 }
}
```

Attack on the DAO

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  if (donations[msg.sender] > 0)
  { msg.sender.call.value(donations[msg.sender])();
    donations[msg.sender] = 0;
  }
 }
}
```

Attack on the DAO

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  if (donations[msg.sender] > 0)
  { msg.sender.call.value(donations[msg.sender])();
    donations[msg.sender] = 0;
  }
 }
}
```

Attack on the DAO

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  if (donations[msg.sender] > 0)
  { msg.sender.call.value(donations[msg.sender])();
    donations[msg.sender] = 0;
  }
 }
}
```

Attack on the DAO

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  if (donations[msg.sender] > 0)
  { msg.sender.call.value(donations[msg.sender])();
    donations[msg.sender] = 0;
  }
 }
}
```

Attack on the DAO

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  if (donations[msg.sender] > 0)
  { msg.sender.call.value(donations[msg.sender])();
    donations[msg.sender] = 0;
  }
 }
}
```

Attack on the DAO

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  if (donations[msg.sender] > 0)
  { msg.sender.call.value(donations[msg.sender])();
    donations[msg.sender] = 0;
  }
 }
}
```

So, what did go wrong here?

Common approch from the literature

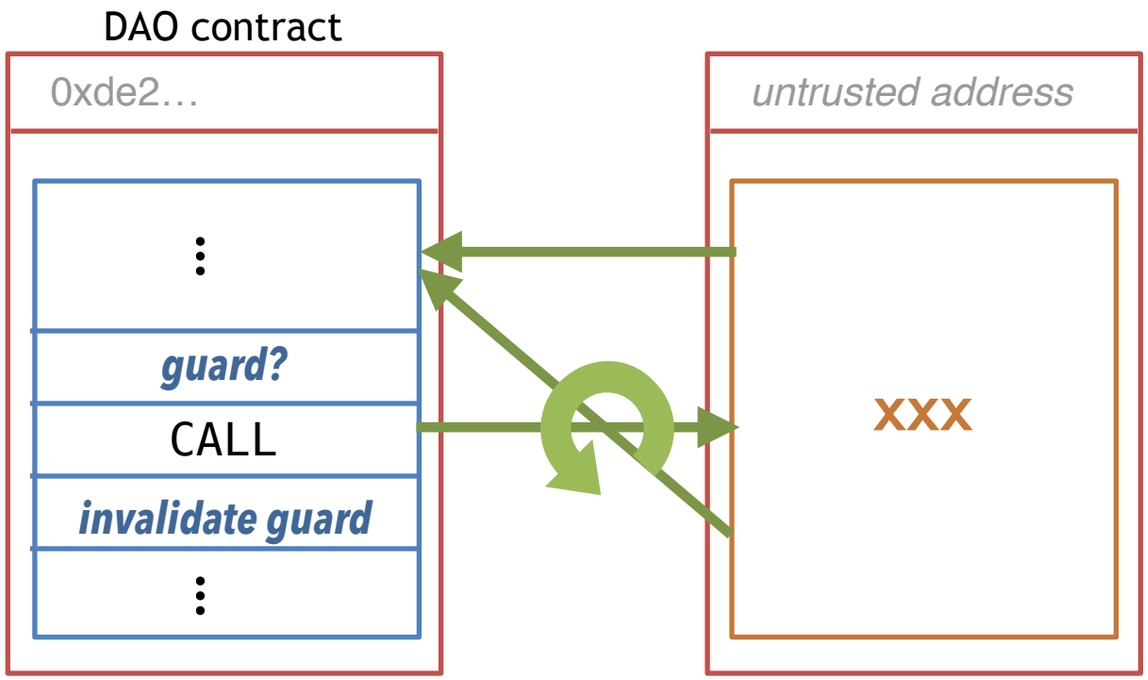
- “the guard should be invalidated before performing the call”
- Syntactic and program specific characterization
- What is the underlying semantic security property?

Call integrity

- Reason for the DAO:
  untrusted contracts could influence the call flow of the contract (Call integrity)

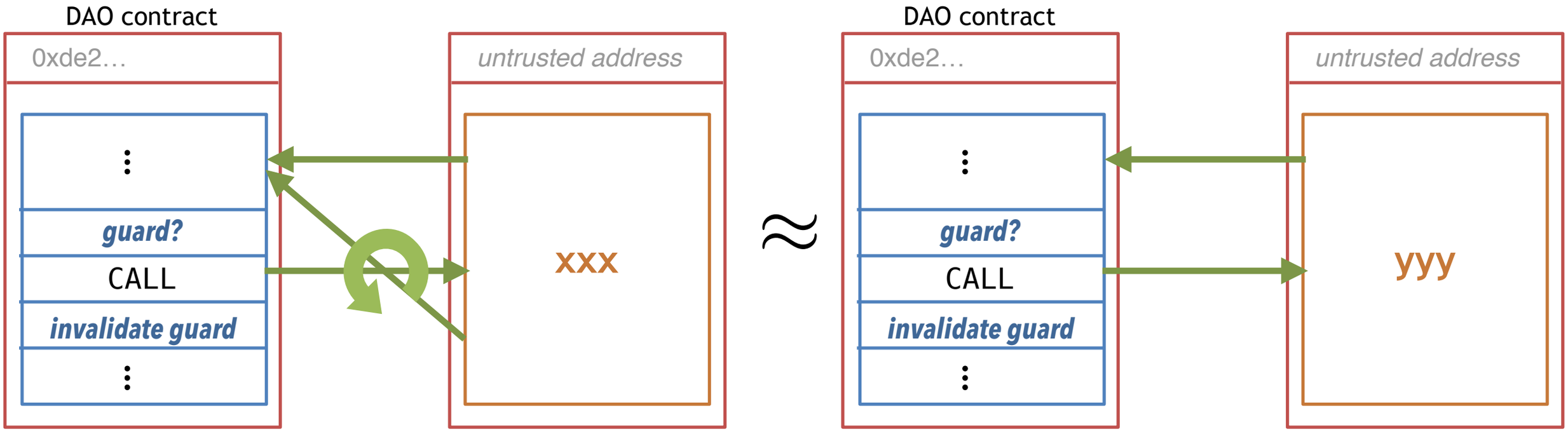
Call integrity

- Reason for the DAO:
  untrusted contracts could influence the call flow of the contract (Call integrity)



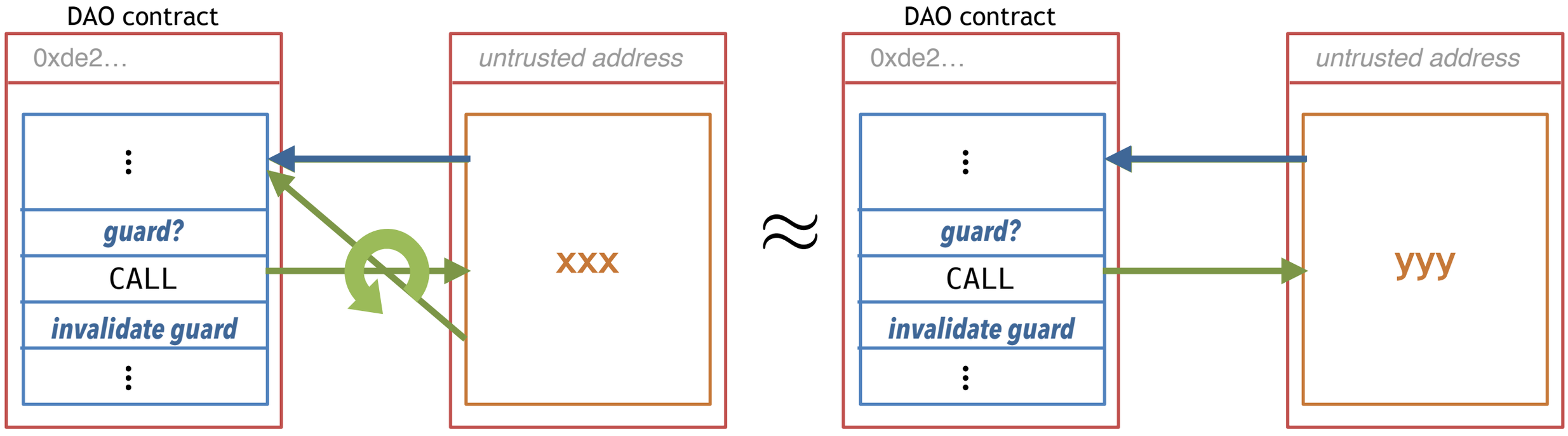
Call integrity

- Reason for the DAO:
  untrusted contracts could influence the call flow of the contract (Call integrity)



Call integrity

- Reason for the DAO:
  untrusted contracts could influence the call flow of the contract (Call integrity)



$$\Gamma \vDash s_c :: S \xrightarrow{\pi}{}^{*} t_c :: S \wedge \mathit{final}(t_c) \wedge \Gamma \vDash {s'}_c :: S' \xrightarrow{\pi'}{}^{*} {t'}_c :: S' \wedge \mathit{final}({t'}_c)$$

$$\Longrightarrow \pi \downarrow_{\mathit{calls}_c} = \pi' \downarrow_{\mathit{calls}_c}$$

Differing only in codes of untrusted addresses
=> c is called in the same way

c should produce the same calls

Hyper-Property

Note: we annotate execution states with the contract (pair of address + code) they are executing:

$S_{(\iota.\mathit{actor},\iota.\mathit{code})}$

Single-entrancy

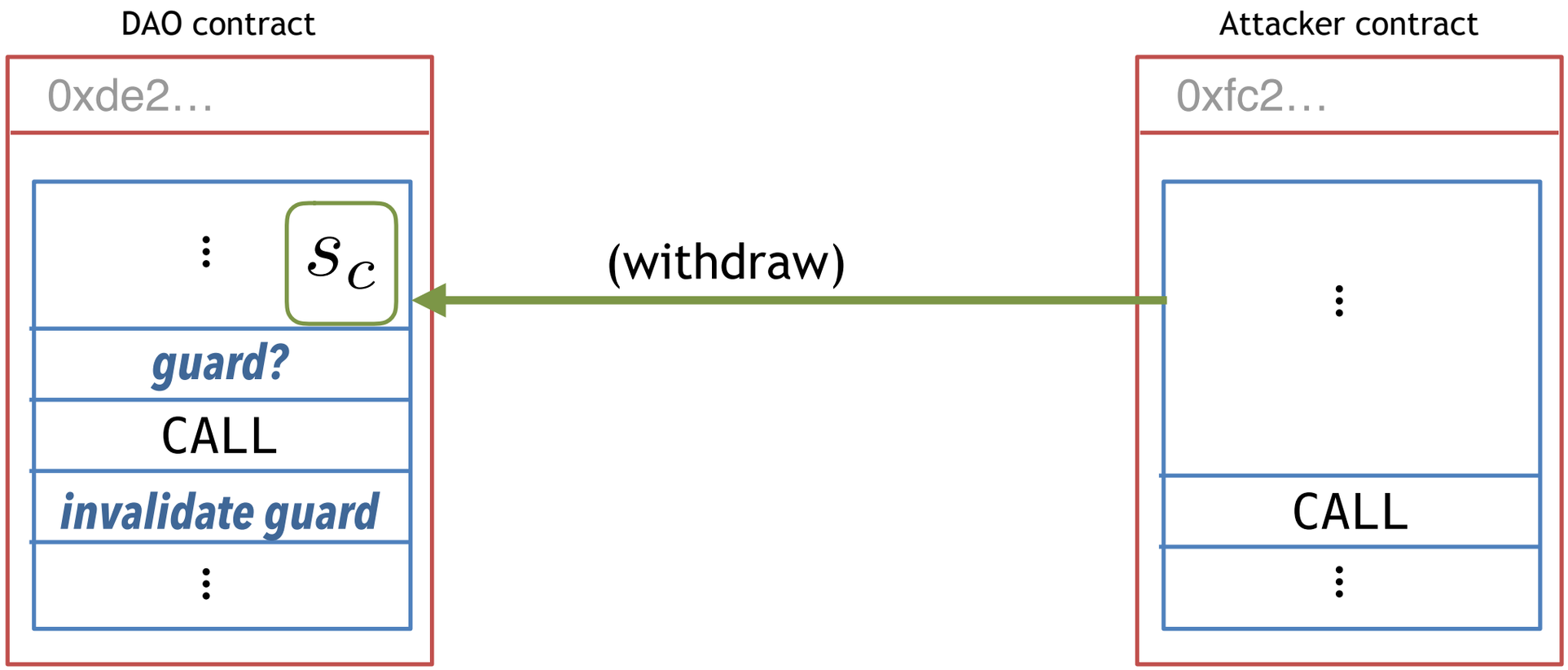
- Single-entrancy for c:
  "After being re-entered, contract c should perform no more calls"

$$\neg\exists s'', c'.\Gamma \vdash s_c :: S \rightarrow^* s''_{c'} :: s'_c :: S' + + s_c :: S$$

Reachability property

Single-entrancy

- Single-entrancy for c:
  "After being re-entered, contract c should perform no more calls"

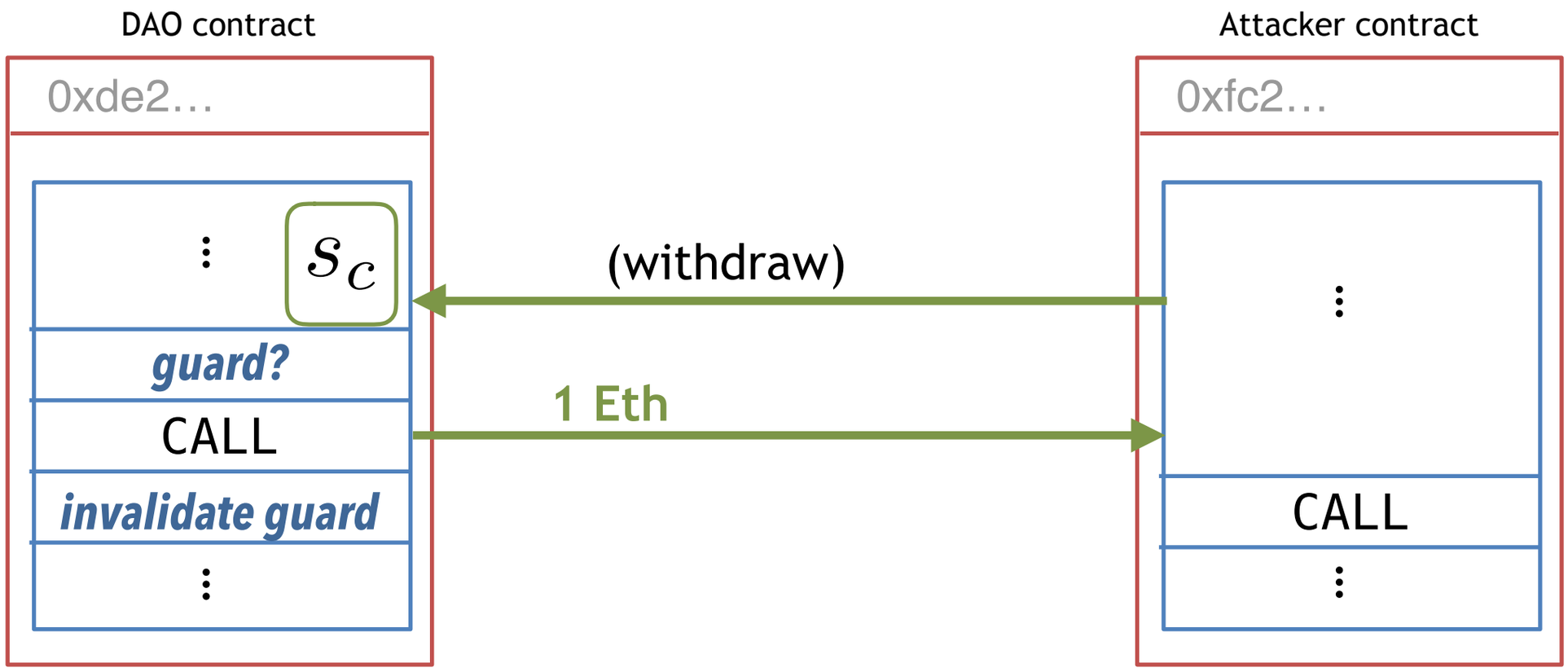


$$\neg\exists s'', c'.\Gamma \vdash s_c :: S \rightarrow^* s''_{c'} :: s'_c :: S' + + s_c :: S$$

Reachability property

Single-entrancy

- Single-entrancy for c:
  "After being re-entered, contract c should perform no more calls"

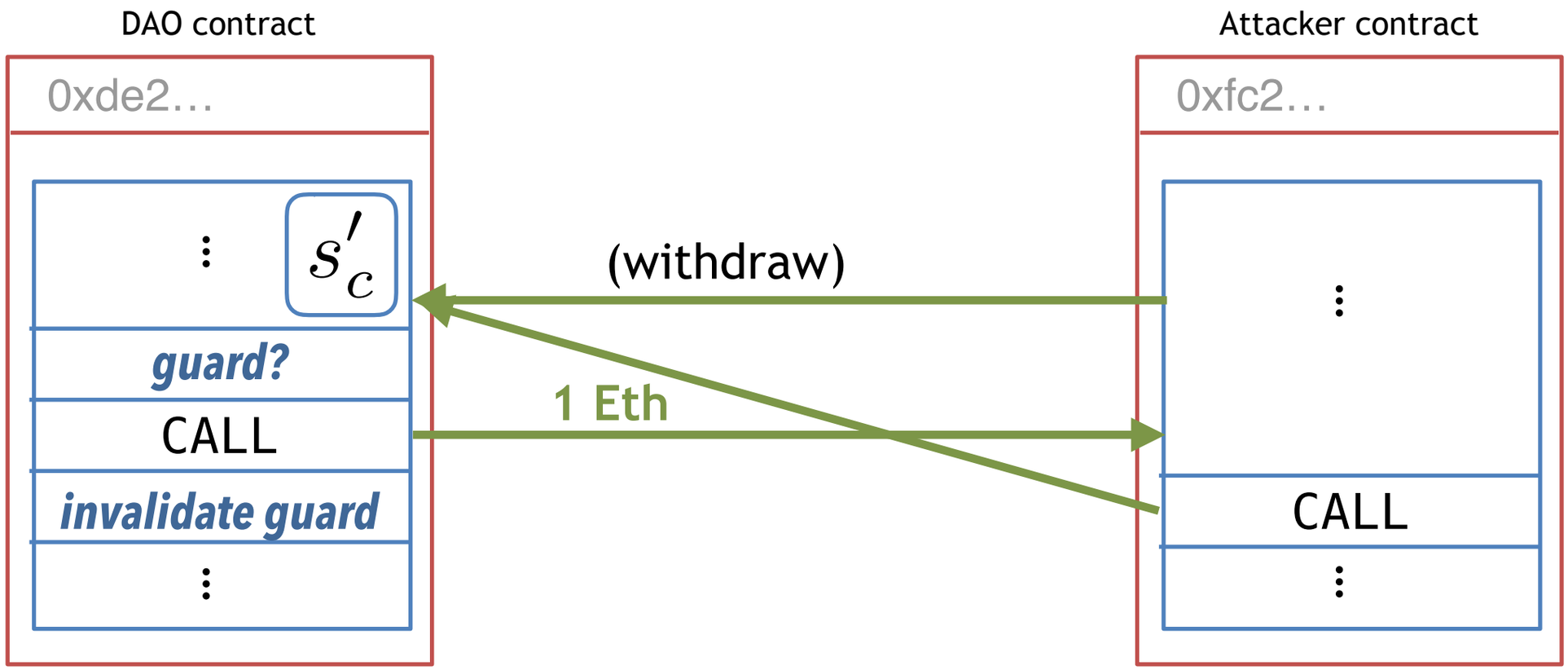


$$\neg\exists s'', c'.\Gamma \vdash s_c :: S \rightarrow^* s''_{c'} :: s'_c :: S' + + s_c :: S$$

Reachability property

Single-entrancy

- Single-entrancy for c:
  "After being re-entered, contract c should perform no more calls"

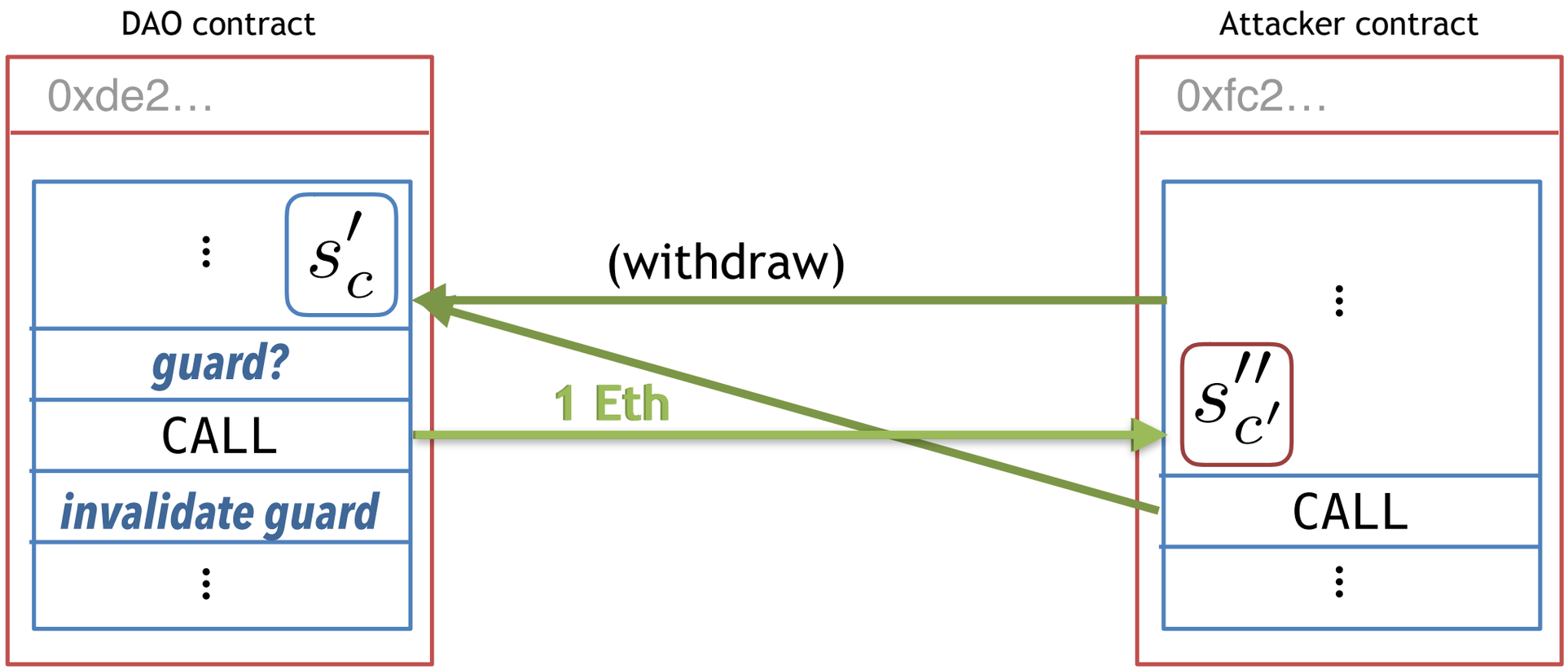


$$\neg\exists s'', c'.\Gamma \vdash s_c :: S \rightarrow^* s''_{c'} :: s'_c :: S' + + s_c :: S$$

Reachability property

Single-entrancy

- Single-entrancy for c:
  "After being re-entered, contract c should perform no more calls"



$$\neg\exists s'', c'.\Gamma \vdash s_c :: S \rightarrow^* s''_{c'} :: s'_c :: S' + + s_c :: S$$

Reachability property

Proof technique for call integrity

Single-entrancy

"contract should not depend on return value of calls to untrusted contracts"

"contract should not depend on untrusted contract's code (that it accesses directly)"

Call Integrity

Proof technique for call integrity

Single-entrancy

"contract should not depend on return value of calls to untrusted contracts"

"contract should not depend on untrusted contract's code (that it accesses directly)"

Call Integrity

Hyper-Property

Hard to verify

Proof technique for call integrity

Reachability Property

Single-entrancy

Value-Dependency Properties

"contract should not depend on return value of calls to untrusted contracts"

"contract should not depend on untrusted contract's code (that it accesses directly)"

Call Integrity

Hyper-Property

Hard to verify

Proof technique for call integrity

Reachability Property

Value-Dependency Properties

Single-entrancy

"contract should not depend on return value of calls to untrusted contracts"

"contract should not depend on untrusted contract's code (that it accesses directly)"

Provable by static analysis: EtherTrust

Call Integrity

Hyper-Property

Hard to verify

Atomicity

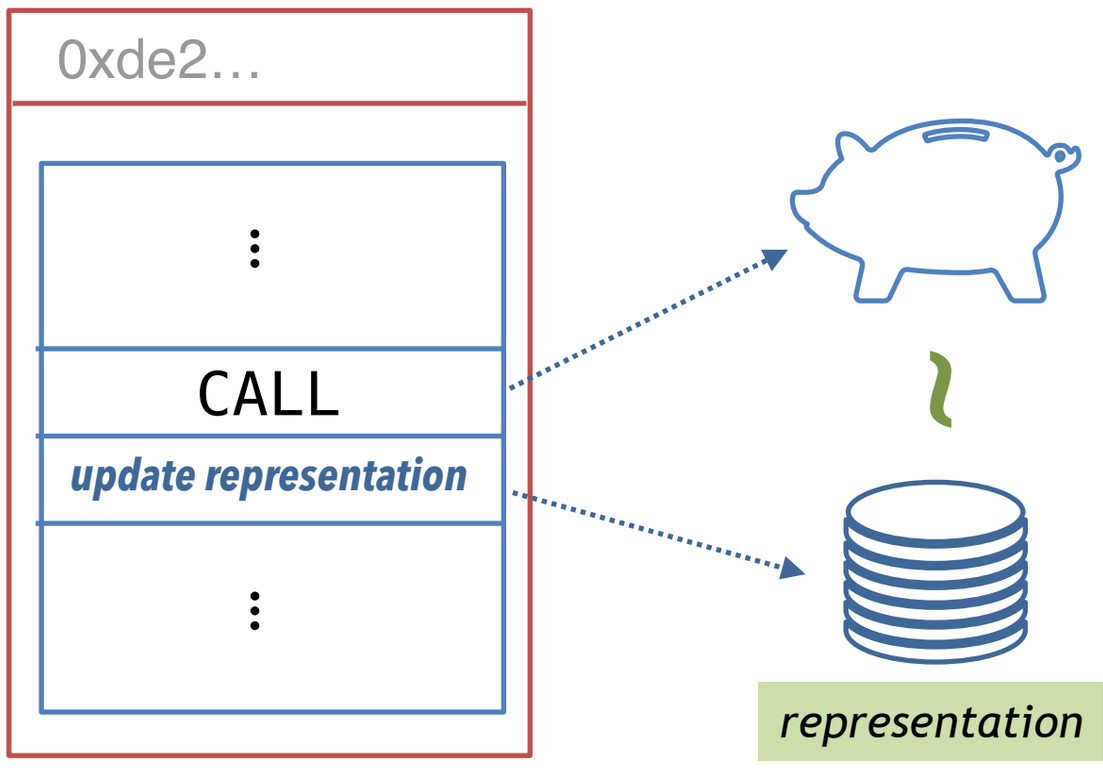
- Inconsistencies due to unhandled gas exceptions

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
   msg.sender.call.value(donations[msg.sender])();
   donations[msg.sender] = 0;
  }
}
```

Atomicity

- Inconsistencies due to unhandled gas exceptions

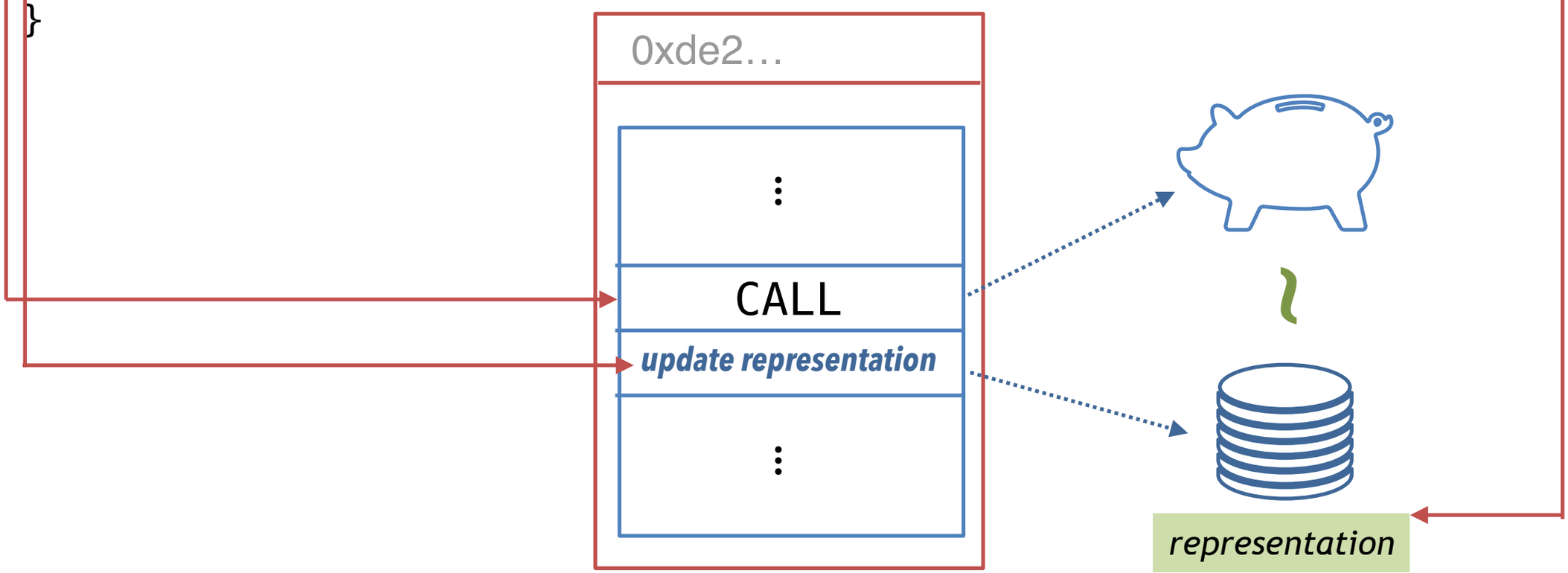
```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
   msg.sender.call.value(donations[msg.sender])();
   donations[msg.sender] = 0;
  }
}
```



Atomicity

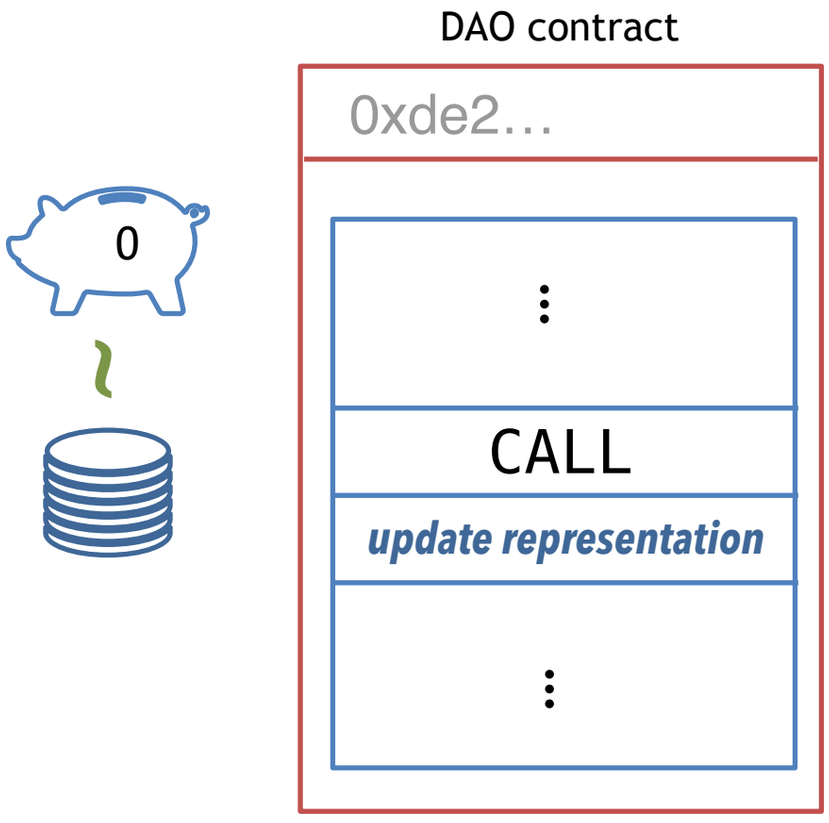
- Inconsistencies due to unhandled gas exceptions

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  msg.sender.call.value(donations[msg.sender])();
  donations[msg.sender] = 0;
 }
}
```



Contract inconsistencies

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  msg.sender.call.value(donations[msg.sender])();
  donations[msg.sender] = 0;
 }
}
```

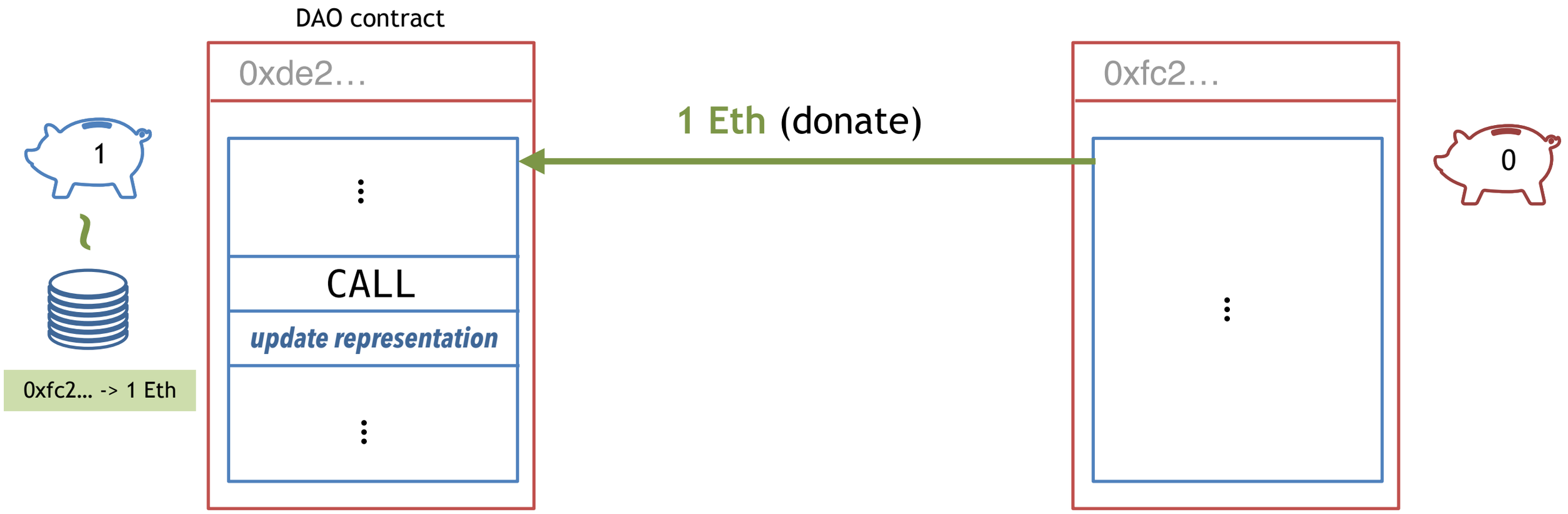


Contract inconsistencies

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  msg.sender.call.value(donations[msg.sender])();
  donations[msg.sender] = 0;
 }
}
```

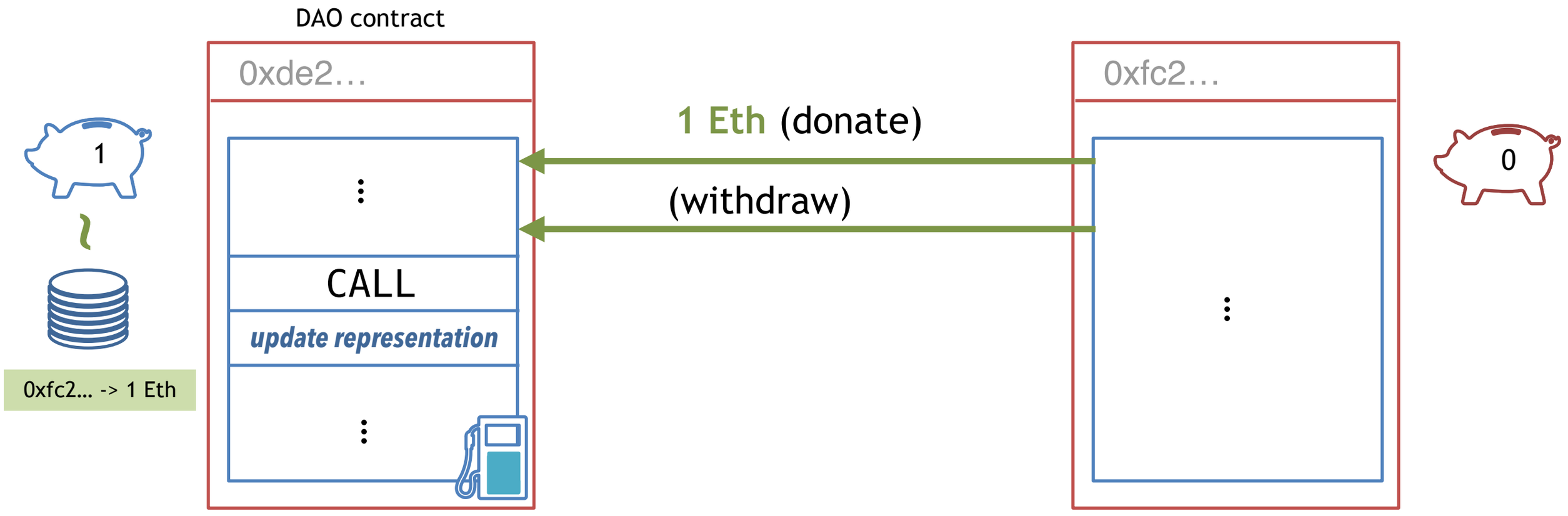
Contract inconsistencies

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  msg.sender.call.value(donations[msg.sender])();
  donations[msg.sender] = 0;
 }
}
```



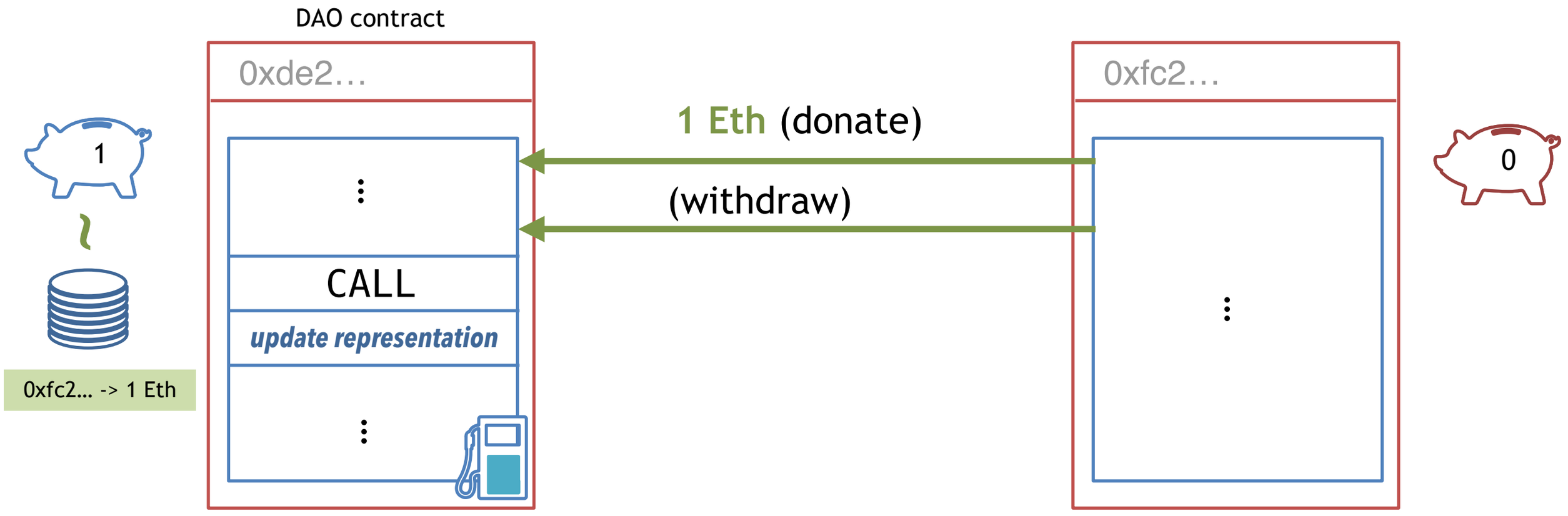
Contract inconsistencies

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  msg.sender.call.value(donations[msg.sender])();
  donations[msg.sender] = 0;
 }
}
```



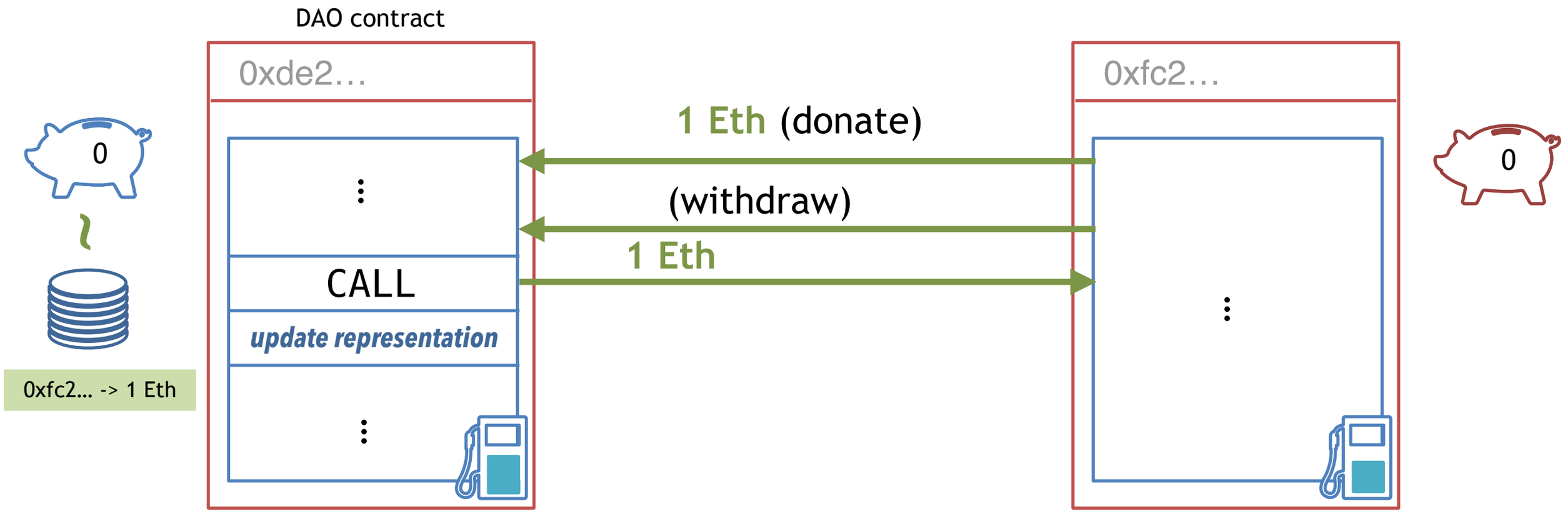
Contract inconsistencies

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  msg.sender.call.value(donations[msg.sender])();
  donations[msg.sender] = 0;
 }
}
```



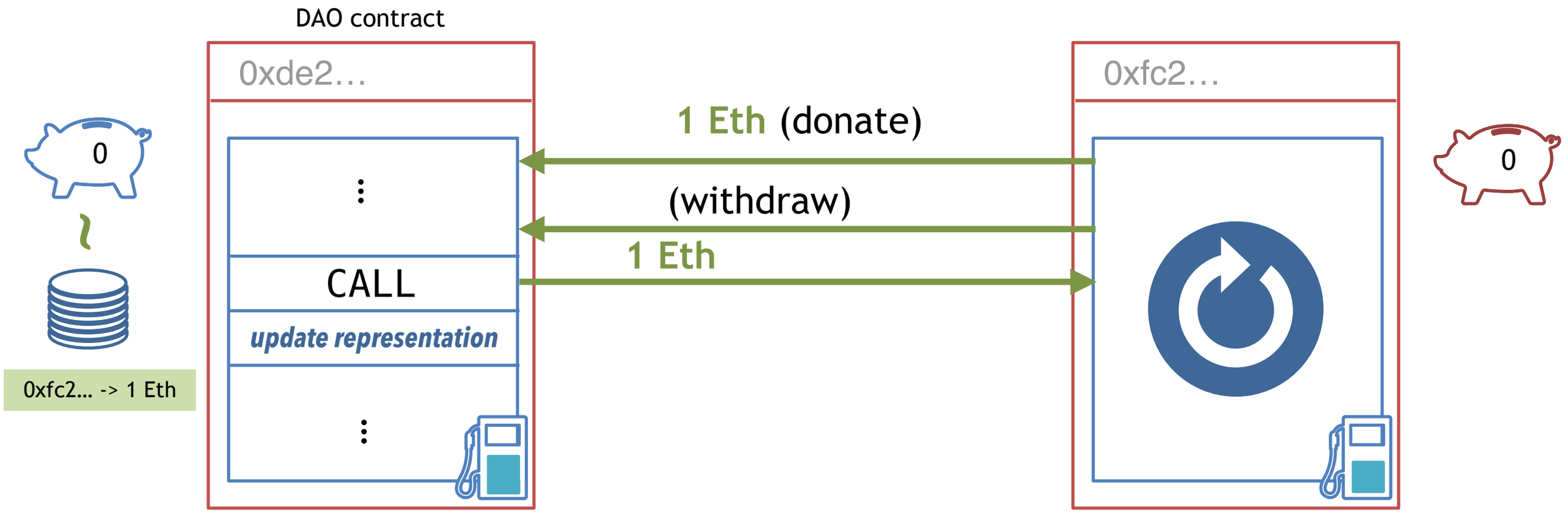
Contract inconsistencies

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  msg.sender.call.value(donations[msg.sender])();
  donations[msg.sender] = 0;
 }
}
```



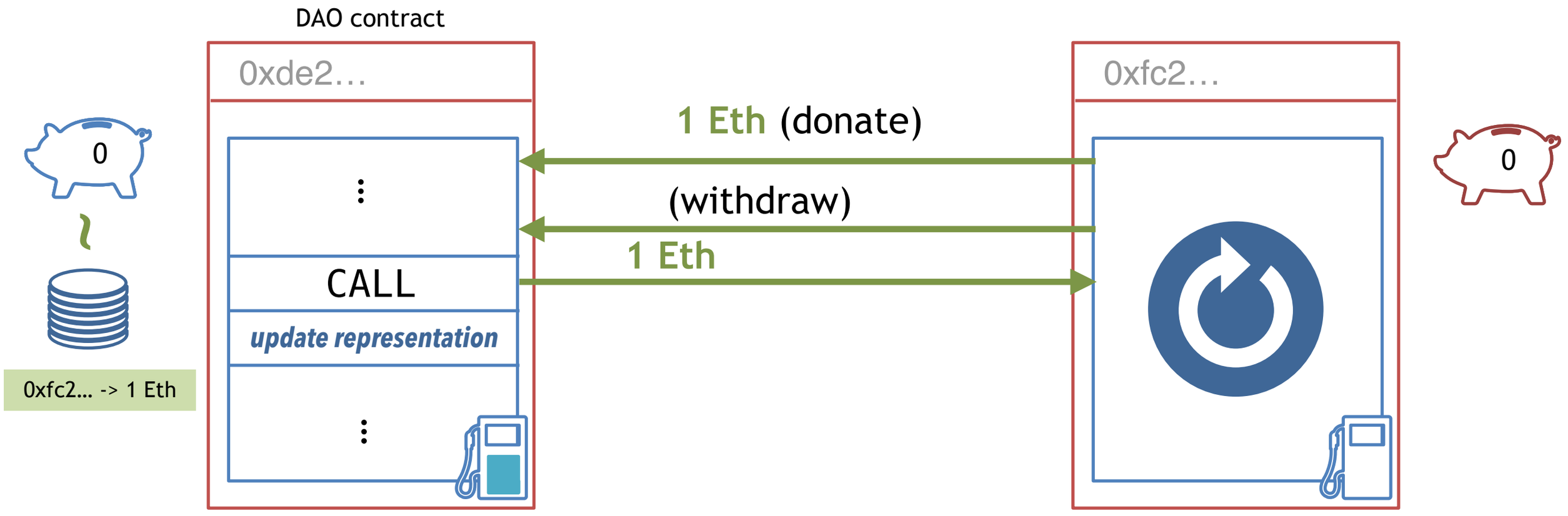
Contract inconsistencies

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  msg.sender.call.value(donations[msg.sender])();
  donations[msg.sender] = 0;
 }
}
```



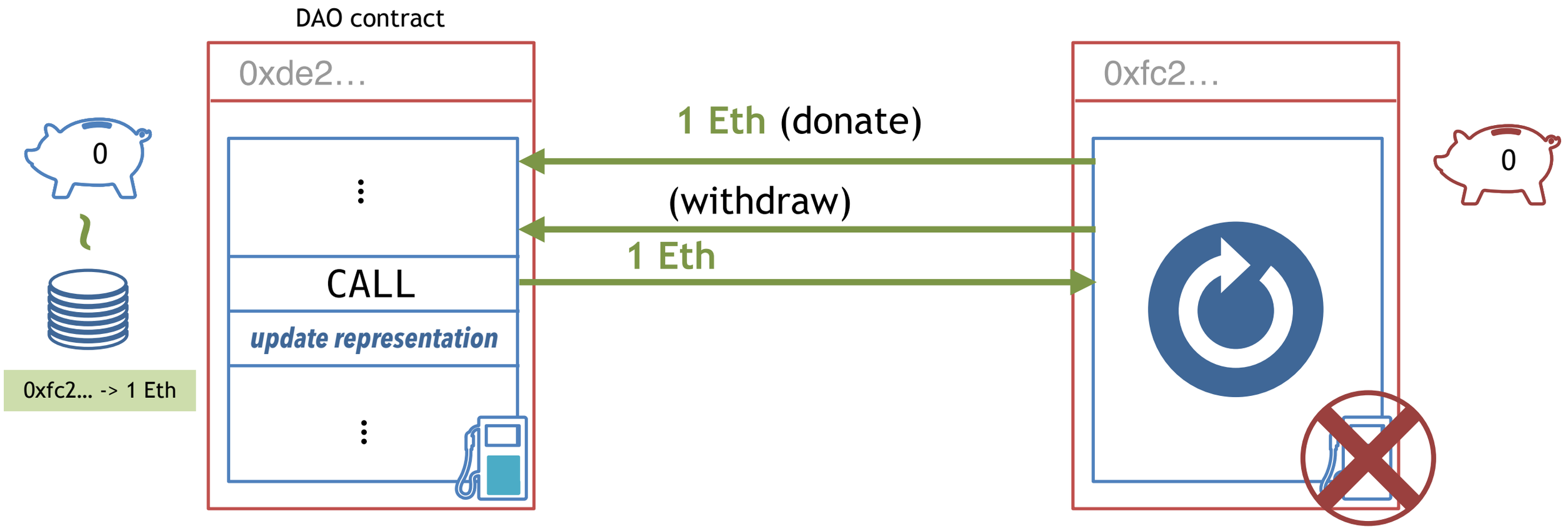
Contract inconsistencies

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  msg.sender.call.value(donations[msg.sender])();
  donations[msg.sender] = 0;
 }
}
```



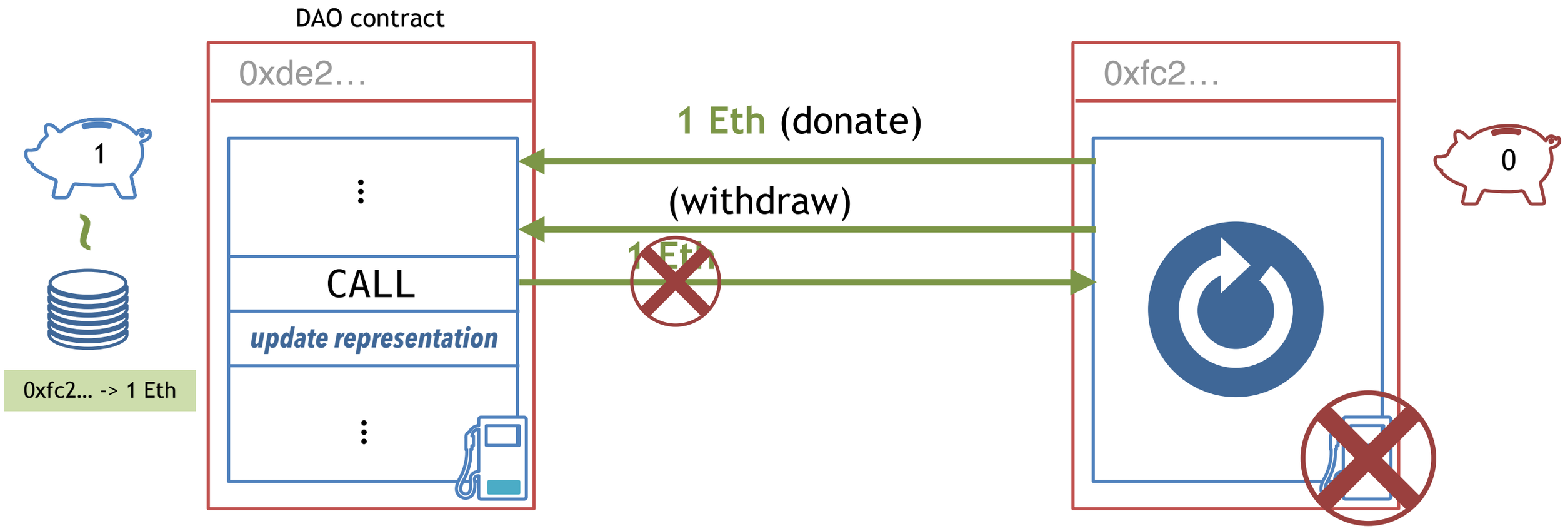
Contract inconsistencies

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  msg.sender.call.value(donations[msg.sender])();
  donations[msg.sender] = 0;
 }
}
```



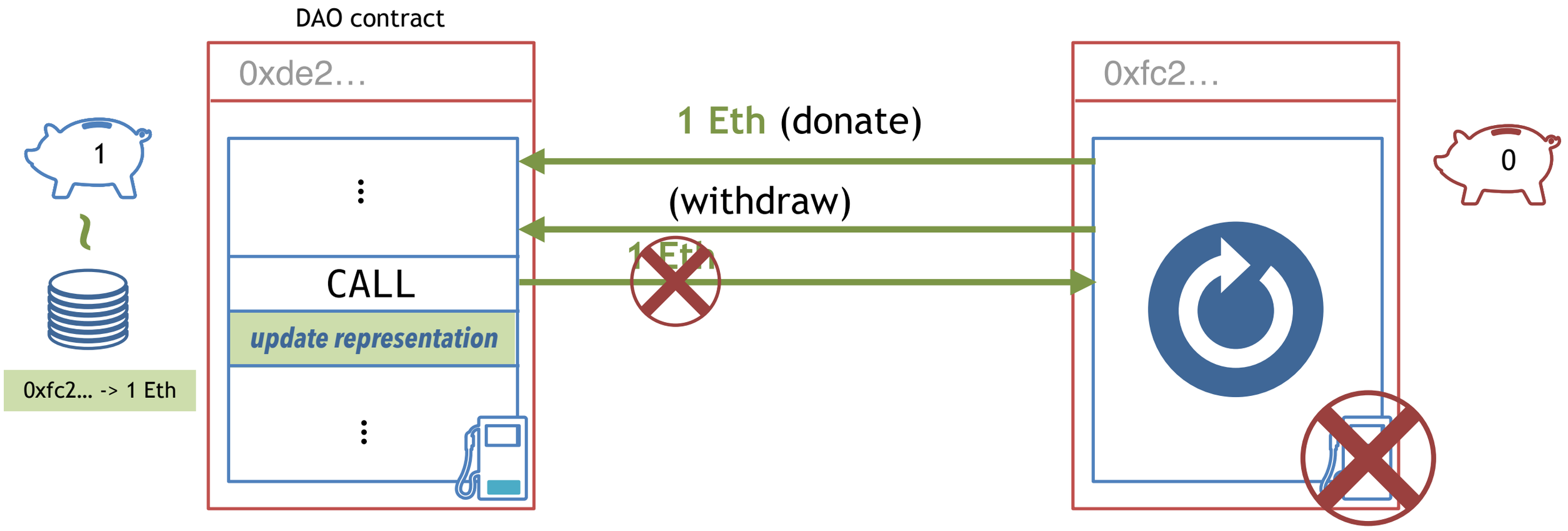
Contract inconsistencies

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  msg.sender.call.value(donations[msg.sender])();
  donations[msg.sender] = 0;
 }
}
```



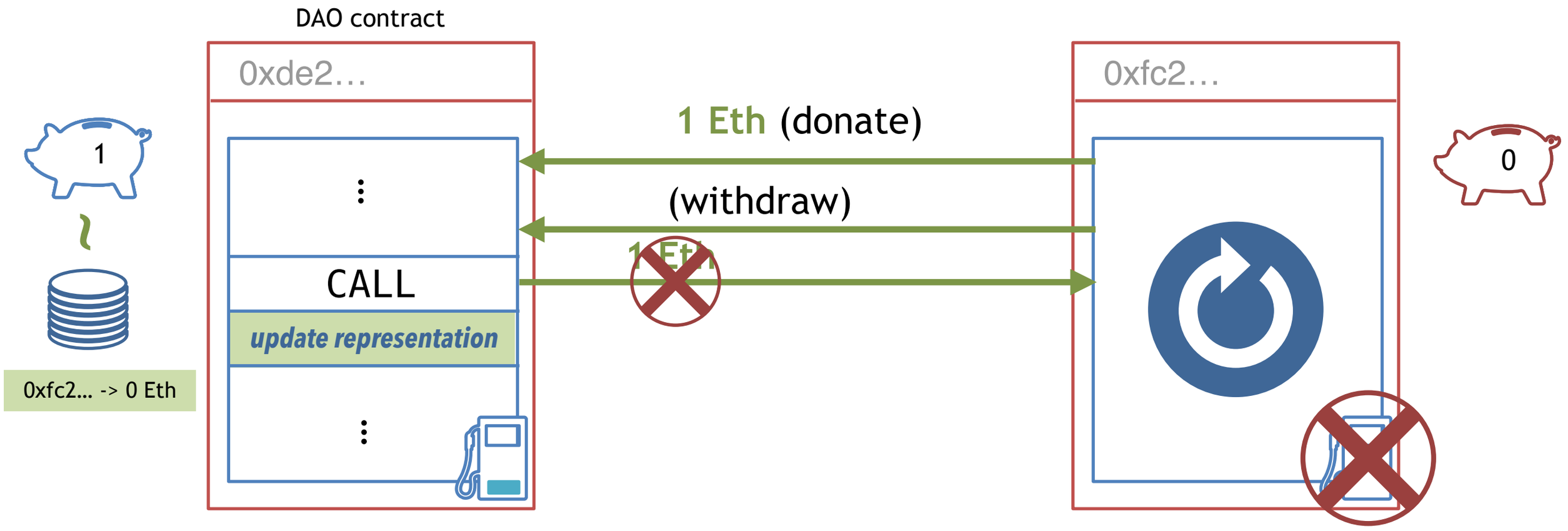
Contract inconsistencies

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  msg.sender.call.value(donations[msg.sender])();
  donations[msg.sender] = 0;
 }
}
```



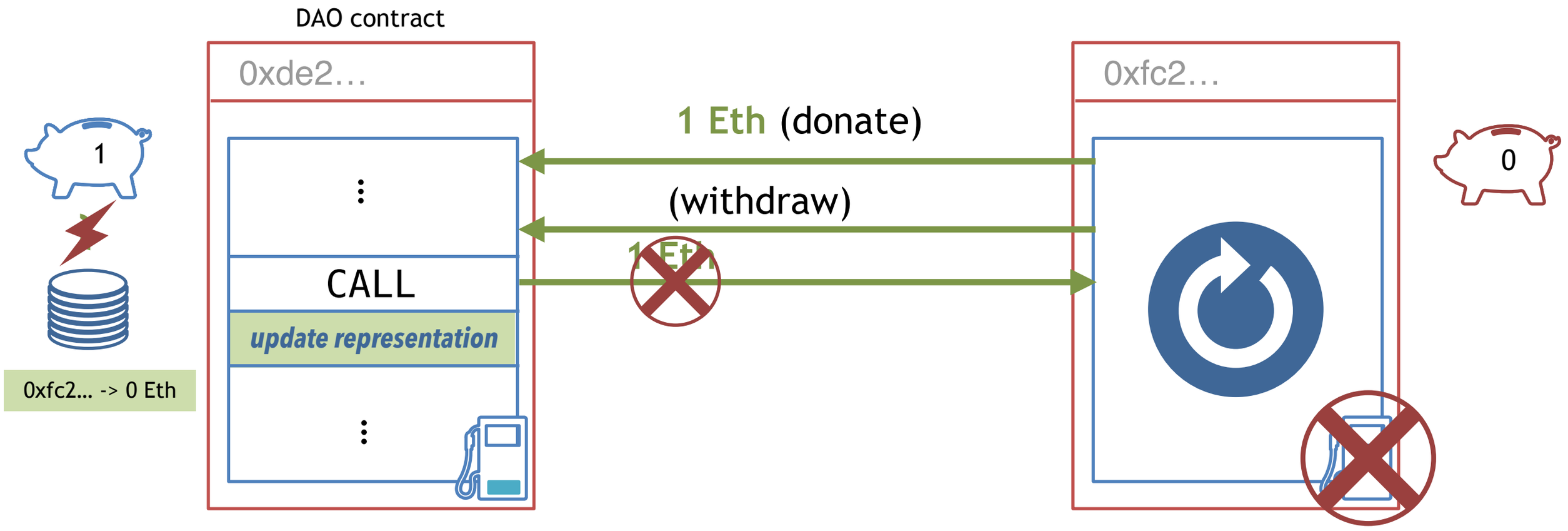
Contract inconsistencies

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  msg.sender.call.value(donations[msg.sender])();
  donations[msg.sender] = 0;
 }
}
```



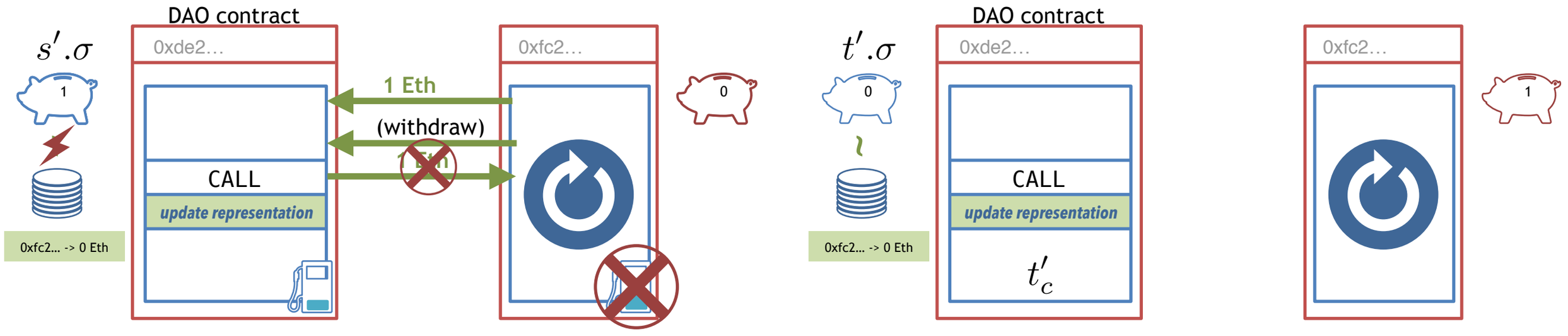
Contract inconsistencies

```
contract DAO {
 mapping (address => uint) donations;

 function donate() {
  donations[msg.sender] += msg.value;
 }

 function withdraw(){
  msg.sender.call.value(donations[msg.sender])();
  donations[msg.sender] = 0;
 }
}
```



Atomicity

- Reason for inconsistency: the (effects of) a contract's execution should not depend on the amount of gas given for execution

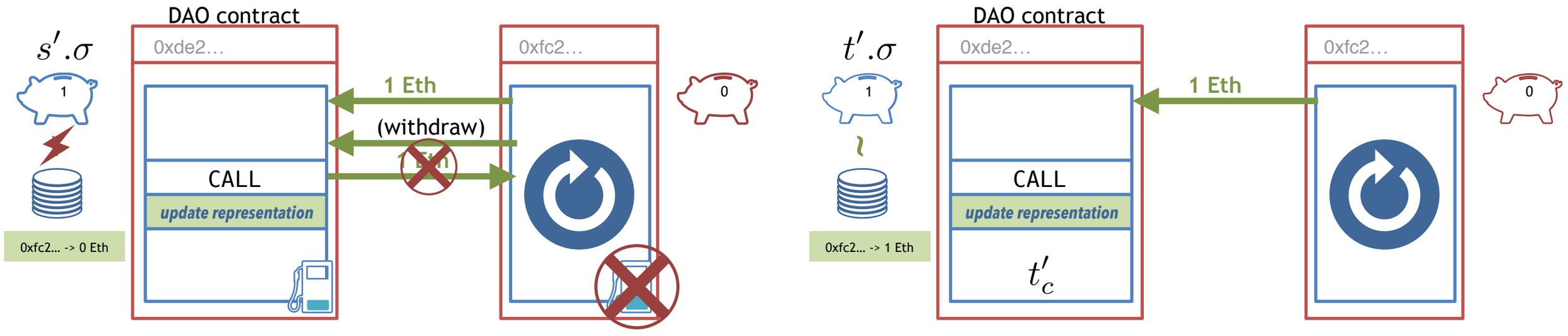


Initial states only differing in gas

$$\Gamma \models s_c :: S \rightarrow^* s'_c :: S \wedge \mathit{final}(s') \wedge \ \Gamma \models t_c :: S \rightarrow^* t'_c :: S \wedge \mathit{final}(t')$$
$$\Rightarrow s'.\sigma = t'.\sigma \vee s.\sigma = s'.\sigma \vee t.\sigma = t'.\sigma$$

Atomicity

- Reason for inconsistency: the (effects of) a contract's execution should not depend on the amount of gas given for execution

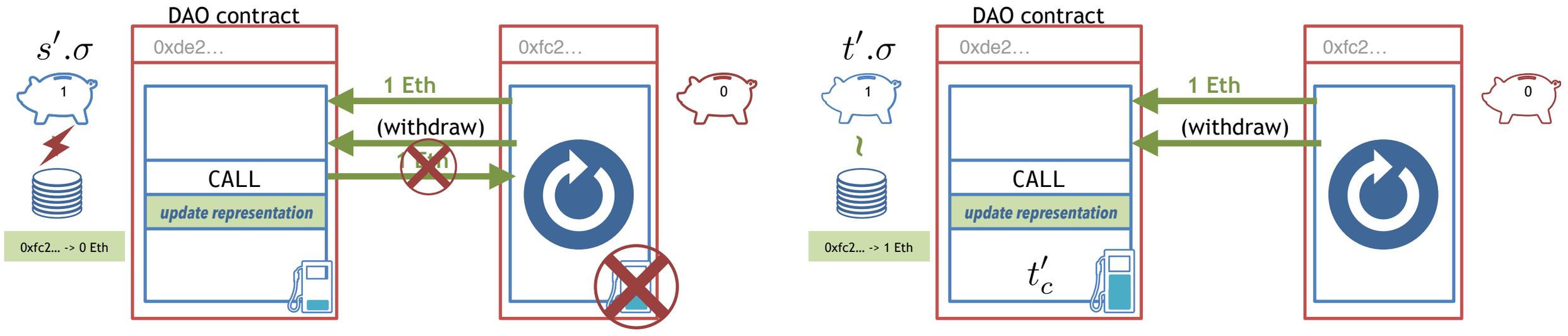


Initial states only differing in gas

$$\Gamma \models s_c :: S \rightarrow^* s'_c :: S \wedge \mathit{final}(s') \wedge \; \Gamma \models t_c :: S \rightarrow^* t'_c :: S \wedge \mathit{final}(t')$$
$$\Rightarrow s'.\sigma = t'.\sigma \vee s.\sigma = s'.\sigma \vee t.\sigma = t'.\sigma$$

Atomicity

- Reason for inconsistency: the (effects of) a contract's execution should not depend on the amount of gas given for execution

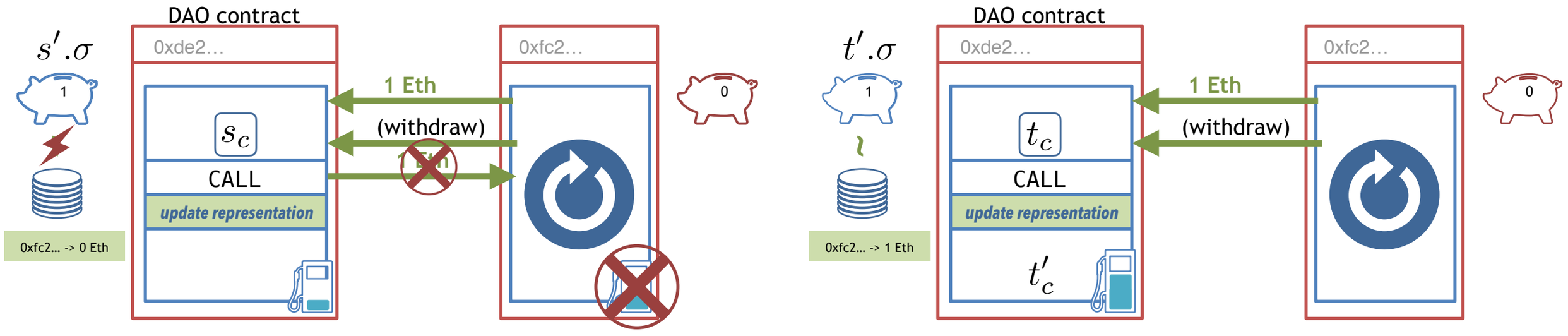


Initial states only differing in gas

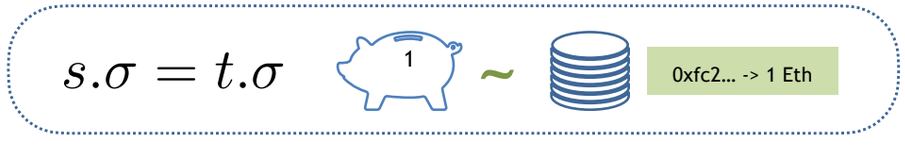
$$\Gamma \models s_c :: S \rightarrow^* s'_c :: S \wedge \mathit{final}(s') \wedge \ \Gamma \models t_c :: S \rightarrow^* t'_c :: S \wedge \mathit{final}(t')$$
$$\Rightarrow s'.\sigma = t'.\sigma \vee s.\sigma = s'.\sigma \vee t.\sigma = t'.\sigma$$

Atomicity

- Reason for inconsistency: the (effects of) a contract's execution should not depend on the amount of gas given for execution



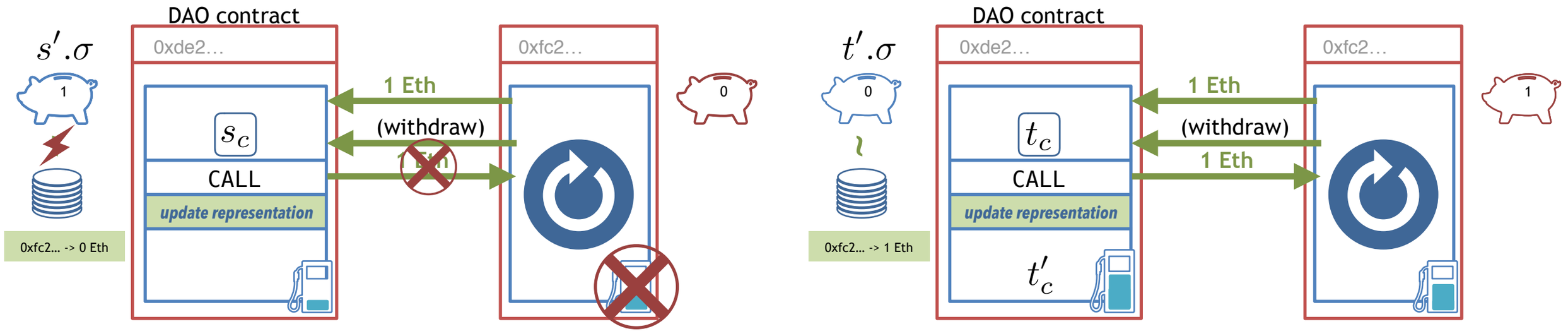
Initial states only differing in gas



$$\Gamma \models \boxed{s_c} :: S \rightarrow^* s'_c :: S \wedge \mathit{final}(s') \wedge \; \Gamma \models \boxed{t_c} :: S \rightarrow^* t'_c :: S \wedge \mathit{final}(t')$$
$$\Rightarrow s'.\sigma = t'.\sigma \vee s.\sigma = s'.\sigma \vee t.\sigma = t'.\sigma$$

Atomicity

- Reason for inconsistency: the (effects of) a contract's execution should not depend on the amount of gas given for execution



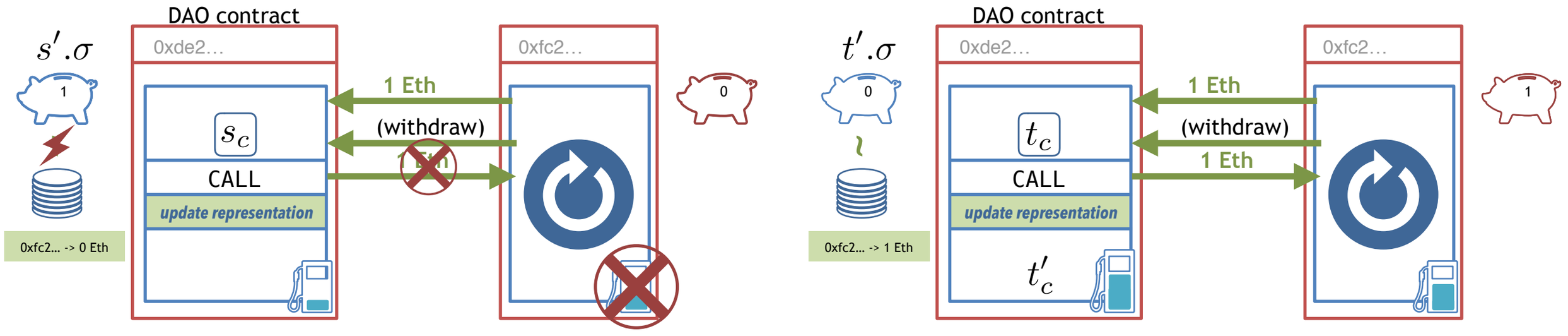
Initial states only differing in gas

$s.\sigma = t.\sigma$ 

$$\Gamma \models \boxed{s_c} :: S \rightarrow^* s'_c :: S \wedge \mathit{final}(s') \wedge \; \Gamma \models \boxed{t_c} :: S \rightarrow^* t'_c :: S \wedge \mathit{final}(t')$$
$$\Rightarrow s'.\sigma = t'.\sigma \vee s.\sigma = s'.\sigma \vee t.\sigma = t'.\sigma$$

Atomicity

- Reason for inconsistency: the (effects of) a contract's execution should not depend on the amount of gas given for execution



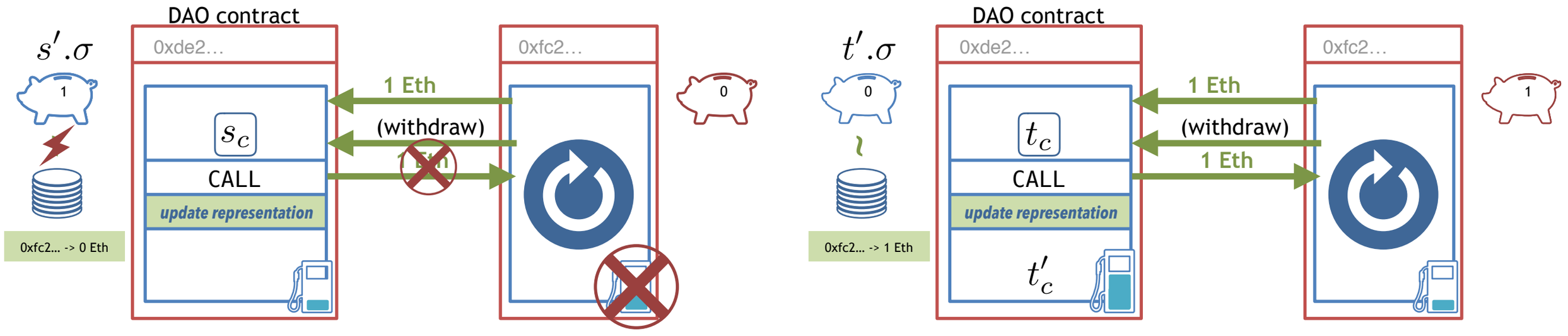
Initial states only differing in gas

$s.\sigma = t.\sigma$ 

$$\Gamma \models \boxed{s_c} :: S \rightarrow^* s'_c :: S \wedge \mathit{final}(s') \wedge \; \Gamma \models \boxed{t_c} :: S \rightarrow^* t'_c :: S \wedge \mathit{final}(t')$$
$$\Rightarrow s'.\sigma = t'.\sigma \vee s.\sigma = s'.\sigma \vee t.\sigma = t'.\sigma$$

Atomicity

- Reason for inconsistency: the (effects of) a contract's execution should not depend on the amount of gas given for execution



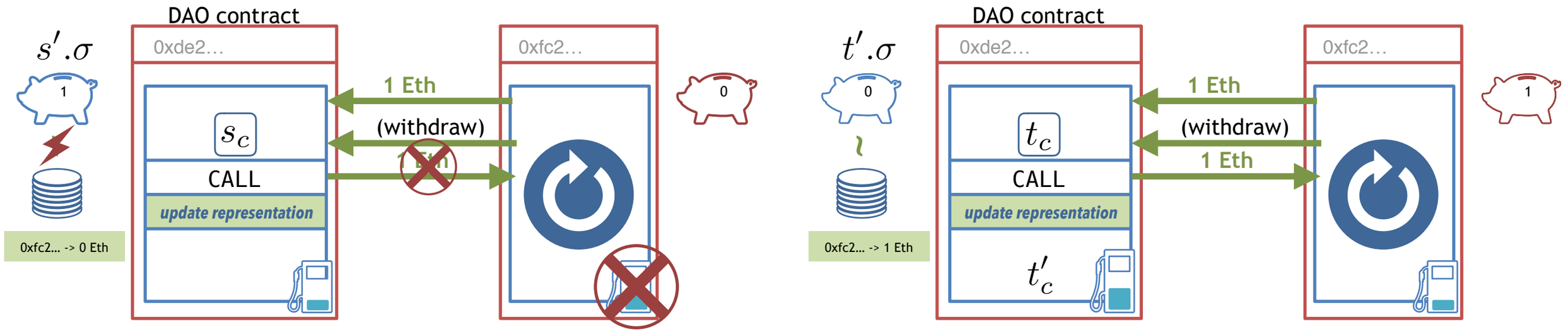
Initial states only differing in gas

$s.\sigma = t.\sigma$ 

$$\Gamma \models \boxed{s_c} :: S \rightarrow^* s'_c :: S \wedge \mathit{final}(s') \wedge \; \Gamma \models \boxed{t_c} :: S \rightarrow^* t'_c :: S \wedge \mathit{final}(t')$$
$$\Rightarrow s'.\sigma = t'.\sigma \vee s.\sigma = s'.\sigma \vee t.\sigma = t'.\sigma$$

Atomicity

- Reason for inconsistency: the (effects of) a contract's execution should not depend on the amount of gas given for execution



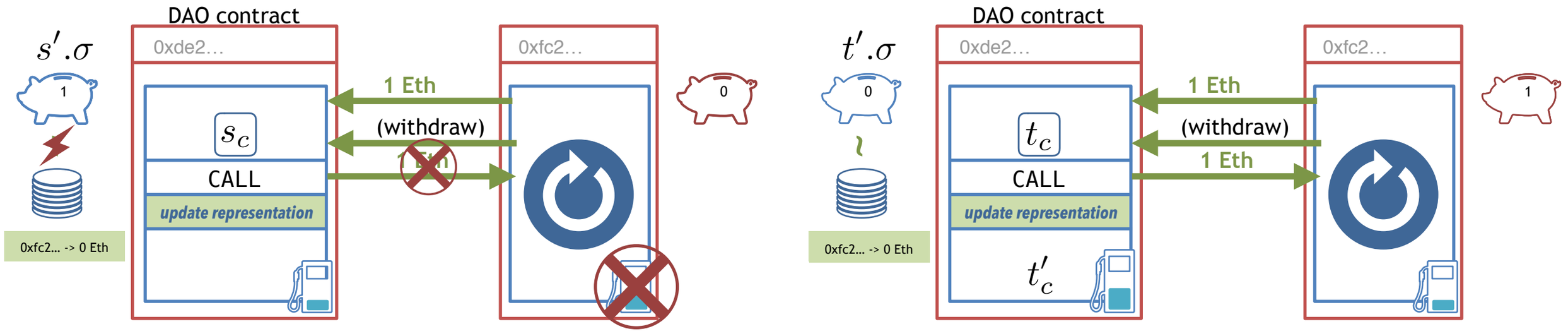
Initial states only differing in gas

$s.\sigma = t.\sigma$ 

$$\Gamma \models \boxed{s_c} :: S \rightarrow^* s'_c :: S \wedge \mathit{final}(s') \wedge \; \Gamma \models \boxed{t_c} :: S \rightarrow^* t'_c :: S \wedge \mathit{final}(t')$$
$$\Rightarrow s'.\sigma = t'.\sigma \vee s.\sigma = s'.\sigma \vee t.\sigma = t'.\sigma$$

Atomicity

- Reason for inconsistency: the (effects of) a contract's execution should not depend on the amount of gas given for execution



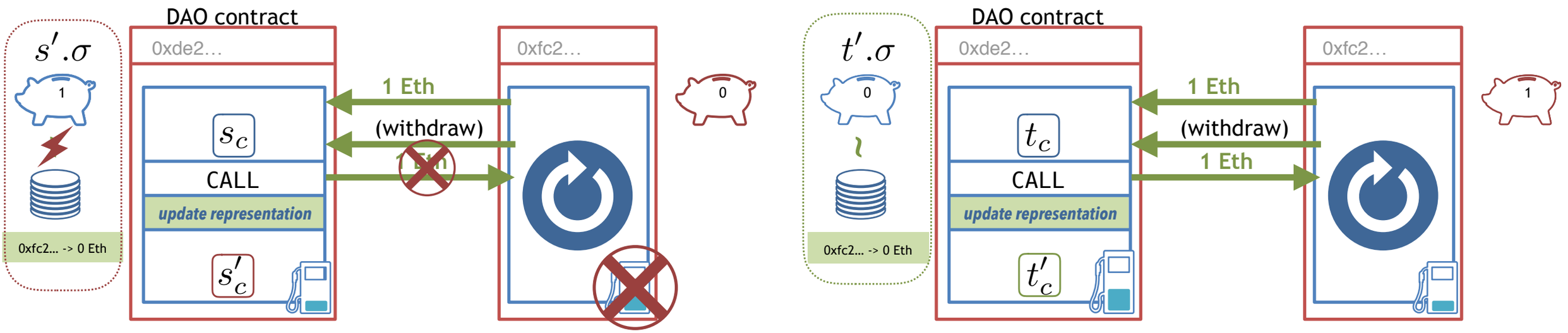
Initial states only differing in gas

$s.\sigma = t.\sigma$ 

$$\Gamma \models \boxed{s_c} :: S \rightarrow^* s'_c :: S \wedge \mathit{final}(s') \wedge \; \Gamma \models \boxed{t_c} :: S \rightarrow^* t'_c :: S \wedge \mathit{final}(t')$$
$$\Rightarrow s'.\sigma = t'.\sigma \vee s.\sigma = s'.\sigma \vee t.\sigma = t'.\sigma$$

Atomicity

- Reason for inconsistency: the (effects of) a contract's execution should not depend on the amount of gas given for execution



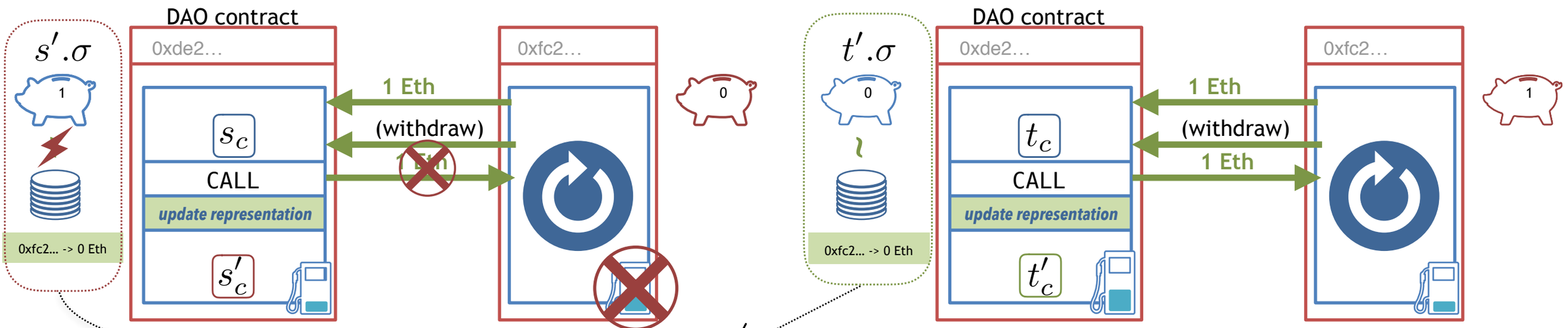
Initial states only differing in gas

$s.\sigma = t.\sigma$ 1 ~ 0xfc2… -> 1 Eth

$$\Gamma \models \boxed{s_c} :: S \rightarrow^* \boxed{s'_c} :: S \wedge final(s') \wedge \ \Gamma \models \boxed{t_c} :: S \rightarrow^* \boxed{t'_c} :: S \wedge final(t')$$
$$\Rightarrow s'.\sigma = t'.\sigma \vee s.\sigma = s'.\sigma \vee t.\sigma = t'.\sigma$$

Atomicity

- Reason for inconsistency: the (effects of) a contract's execution should not depend on the amount of gas given for execution



$$\Gamma \models \boxed{s_c} :: S \rightarrow^* \boxed{s'_c} :: S \wedge final(s') \wedge \ \Gamma \models \boxed{t_c} :: S \rightarrow^* \boxed{t'_c} :: S \wedge final(t')$$
$$\Rightarrow s'.\sigma = t'.\sigma \vee s.\sigma = s'.\sigma \vee t.\sigma = t'.\sigma$$

How can all of that be checked automatically?

Outline

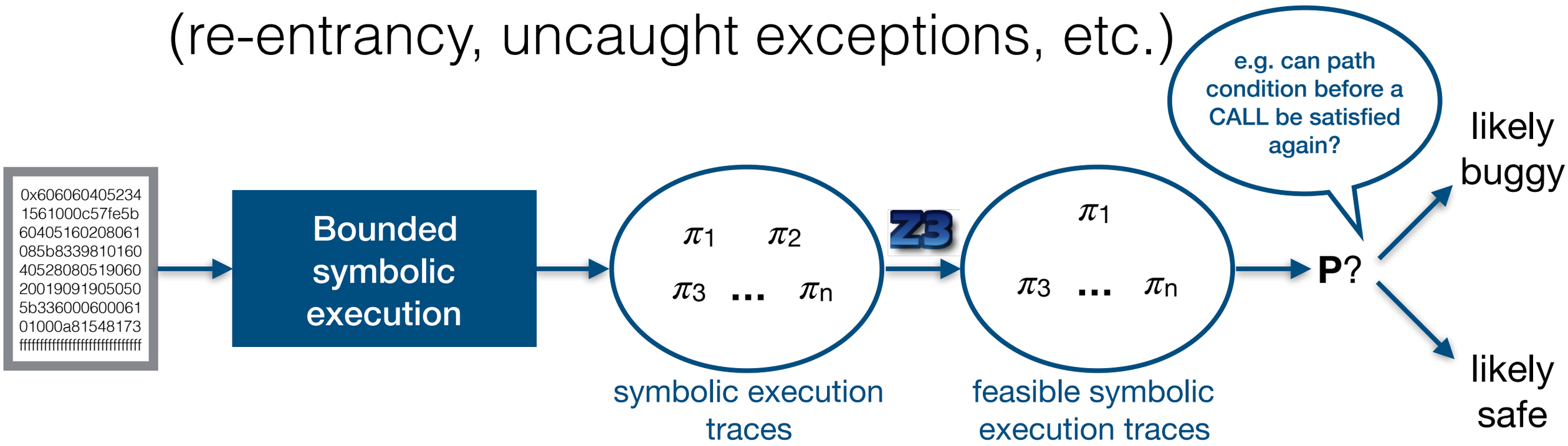
Introduction to Ethereum

Semantics of EVM bytecode

Static Analysis of EVM bytecode

Oyente[1]

- Tool for finding common smart contract bugs in EVM bytecode
  (re-entrancy, uncaught exceptions, etc.)



- Evaluated on ~19000 real world contracts
  (low false positive rate: 6,4%)

[1] Luu, Loi, et al. "Making smart contracts smarter." *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.

Oyente

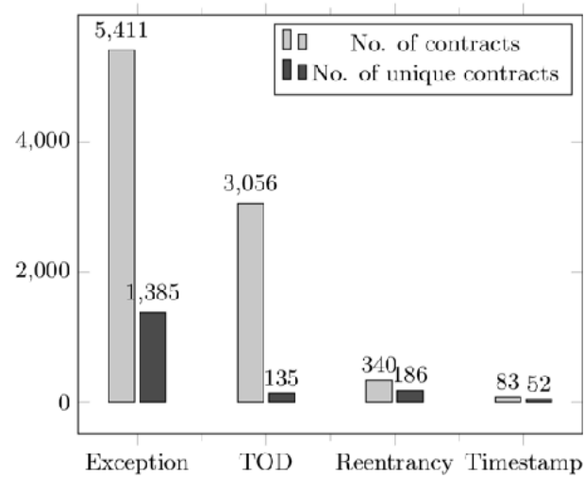


Figure 12: Number of buggy contracts per each security problem reported by OYENTE.

- **Pros**
  - Fast + scalable
- **Cons**
  - Only works for pre-defined properties
  - Produces false positives + false negatives
  - Based on flawed semantics: $\langle S, \sigma \rangle \rightarrow \langle S', \sigma' \rangle$

global state is assumed to be monotonically updated (never reverted) during execution

Oyente

- **Pros**

  Fully Automated

  - Fast + scalable

- **Cons**
  - Only works for pre-defined properties
  - Produces false positives + false negatives
  - Based on flawed semantics: $\langle S, \sigma \rangle \rightarrow \langle S', \sigma' \rangle$

    global state is assumed to be monotonically updated (never reverted) during execution

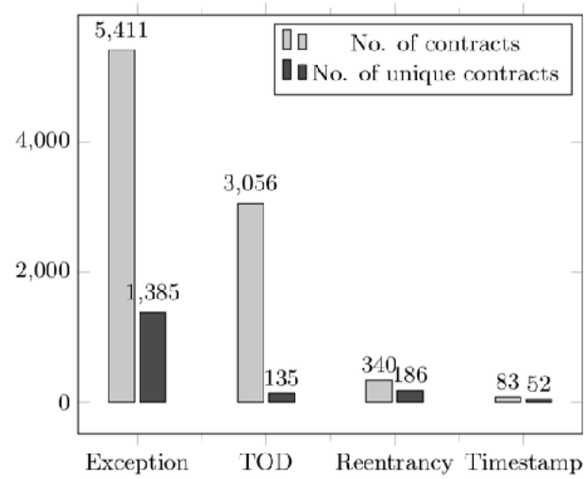


Figure 12: Number of buggy contracts per each security problem reported by OYENTE.

Oyente

- **Pros**

  Fully Automated

  - Fast + scalable

- **Cons**

  Not sound

  - Only works for pre-defined properties
  - Produces false positives + false negatives
  - Based on flawed semantics: $\langle S, \sigma \rangle \rightarrow \langle S', \sigma' \rangle$

    global state is assumed to be monotonically updated (never reverted) during execution

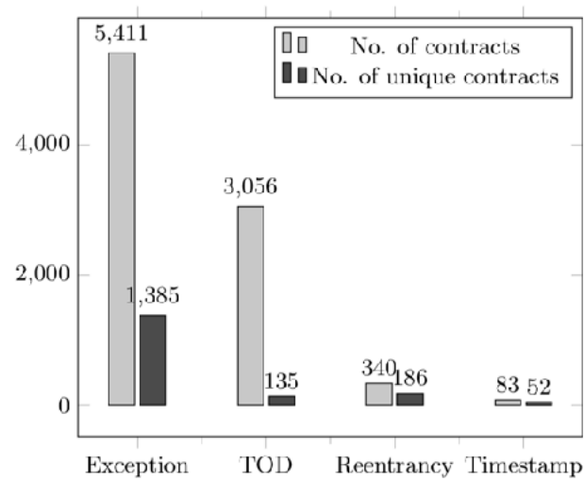


Figure 12: Number of buggy contracts per each security problem reported by OYENTE.

KEVM[2]

- Implementation of EVM bytecode semantics is in the $\mathbb{K}$ framework (rewrite-based executable semantic framework)
- Analysis tools automatically derived from the semantics:
  - Semantic Debugger
  - Program Verifier (for reachability claims)

[2] Hildenbrandt, Everett, et al. *Kevm: A complete semantics of the ethereum virtual machine*. CSF 2018.

KEVM

- **Pros**
  - Based on fully fledged (and tested) semantics of EVM bytecode
  - Allows for Hoare-style-like reasoning
- **Cons**
  - Analysis tool requires the user to specify invariants (semi-automated)
  - No domain-specific over-approximations (e.g. for calling unknown contracts)

KEVM

- **Pros**

Provably sound

  - Based on fully fledged (and tested) semantics of EVM bytecode
  - Allows for Hoare-style-like reasoning

- **Cons**

  - Analysis tool requires the user to specify invariants (semi-automated)
  - No domain-specific over-approximations (e.g. for calling unknown contracts)

KEVM

- **Pros**

Provably sound

  - Based on fully fledged (and tested) semantics of EVM bytecode
  - Allows for Hoare-style-like reasoning

- **Cons**

Only semi-automated

  - Analysis tool requires the user to specify invariants (semi-automated)
  - No domain-specific over-approximations (e.g. for calling unknown contracts)

KEVM

- **Pros**

Sound by construction

  - Based on fully fledged (and tested) semantics of EVM bytecode
  - Allows for Hoare-style-like reasoning

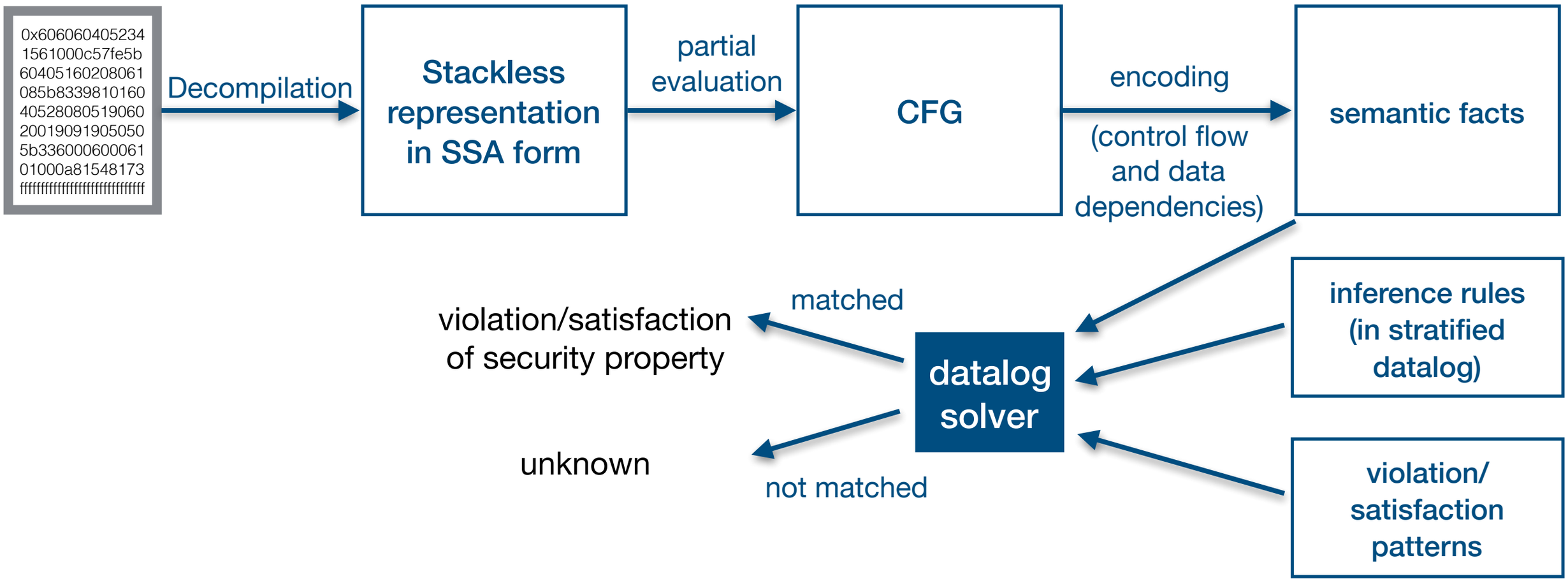
- **Cons**

Only semi-automated

  - Analysis tool requires the user to specify invariants (semi-automated)
  - No domain-specific over-approximations (e.g. for calling unknown contracts)

Securify[3]

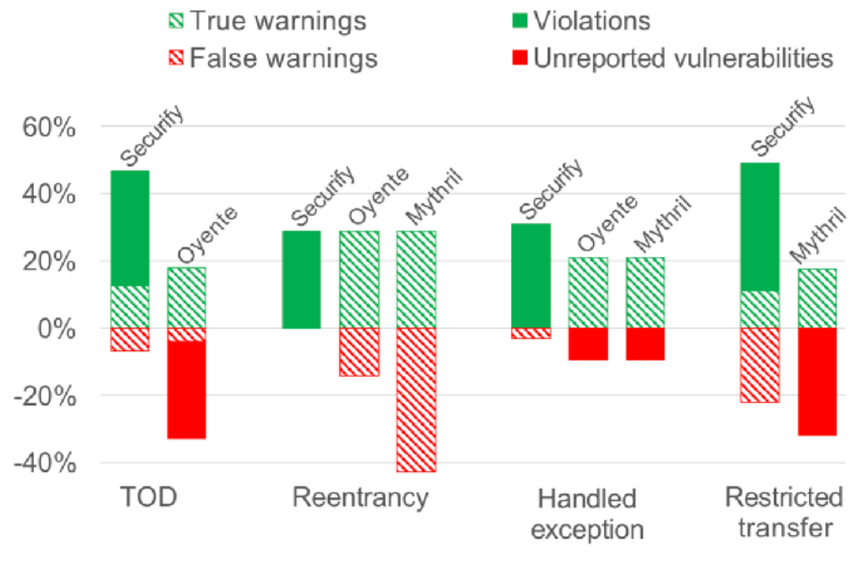
- Static smart contract analyser for EVM bytecode based on ‘semantic fact checking’



- Evaluated on ~25000 real world contracts

[3] Tsankov, Petar, et al. "Securify: Practical Security Analysis of Smart Contracts." arXiv preprint arXiv:1806.01143 (2018).

Securify



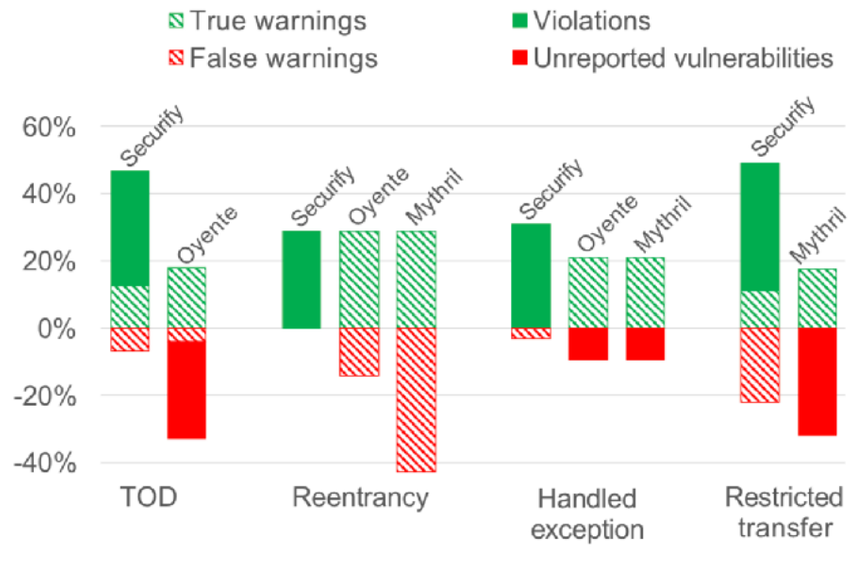
- **Pros**
  - Fast + scalable
  - Shows good accuracy thanks to classification into (confirmed) violations and compliances
- **Cons**
  - Decompilation is not guaranteed to succeed
  - No soundness proof (neither for the dependency analysis nor for the security patterns)

Securify

- **Pros**

Fully Automated

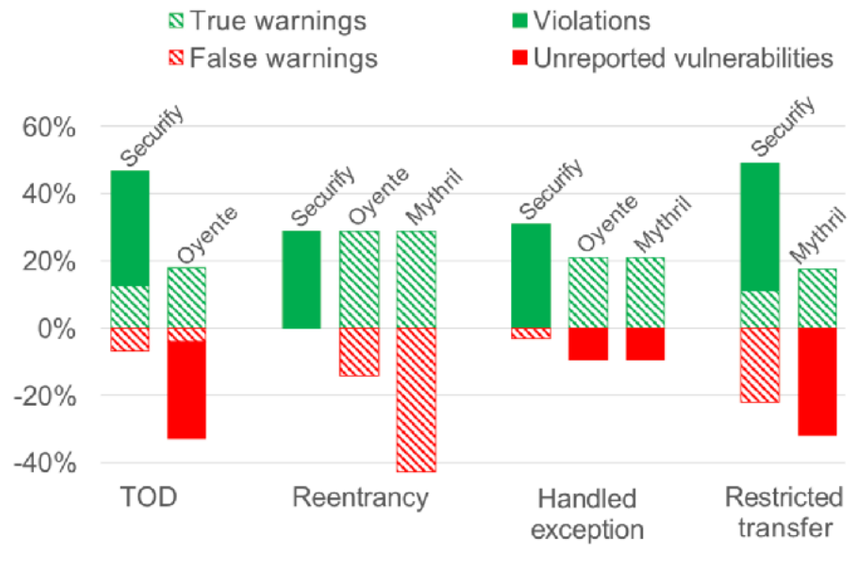
  - Fast + scalable
  - Shows good accuracy thanks to classification into (confirmed) violations and compliances



- **Cons**

  - Decompilation is not guaranteed to succeed
  - No soundness proof (neither for the dependency analysis nor for the security patterns)

Securify



- **Pros**

Fully Automated

  - Fast + scalable
  - Shows good accuracy thanks to classification into (confirmed) violations and compliances

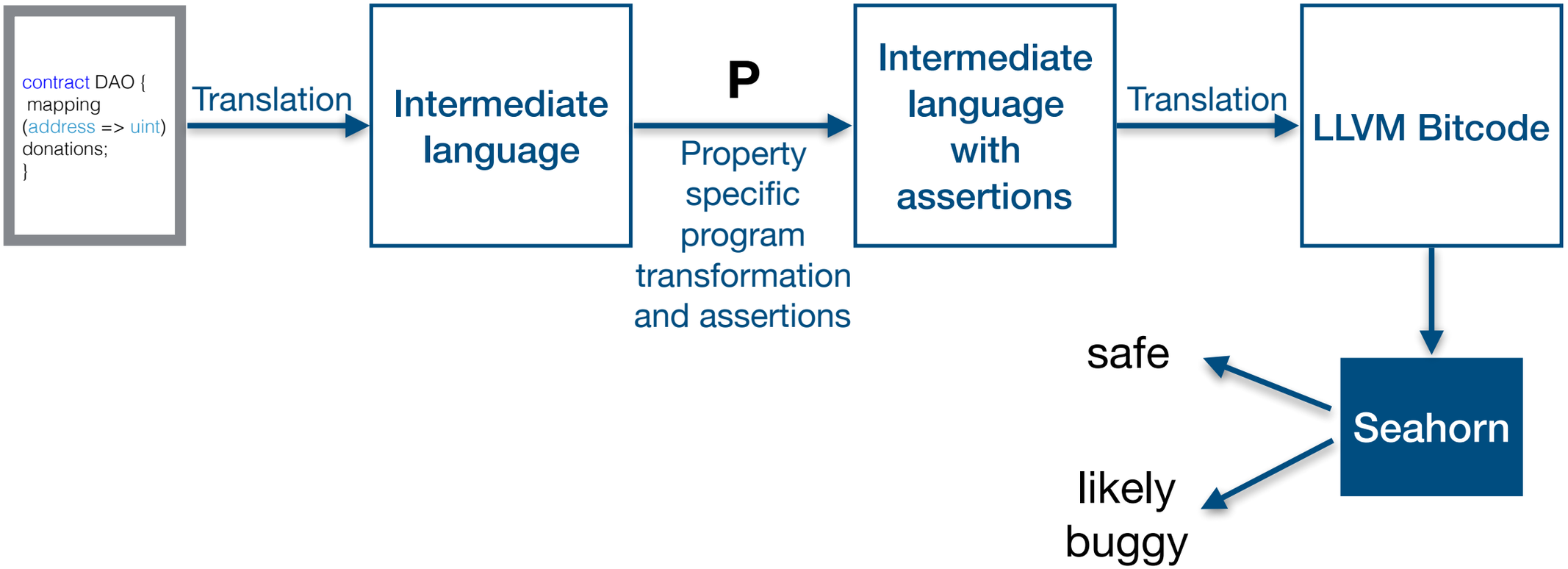
- **Cons**

No soundness proof

  - Decompilation is not guaranteed to succeed
  - No soundness proof (neither for the dependency analysis nor for the security patterns)

ZEUS[4]

- Static analyser for Solidity code

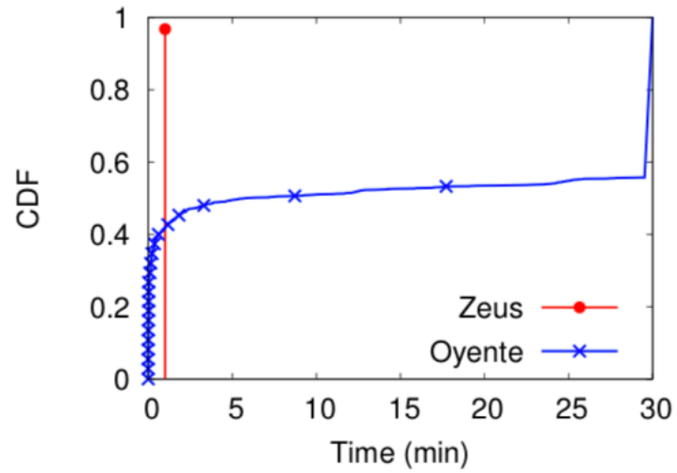


- Evaluated on ~22500 real-world contracts

[4] Kalra, Sukrit, et al. "Zeus: Analyzing safety of smart contracts." NDSS, 2018.

ZEUS

- **Pros**
  - Fast + scalable
- **Cons**
  - Only works on Solidity code (not on bytecode)
  - Only works for pre-defined properties
  - Does not give soundness guarantees (transformations are not semantics preserving + security invariants are not proven sound)
  - Based (as Oyente) on flawed semantics



(d) Verification time in minutes.

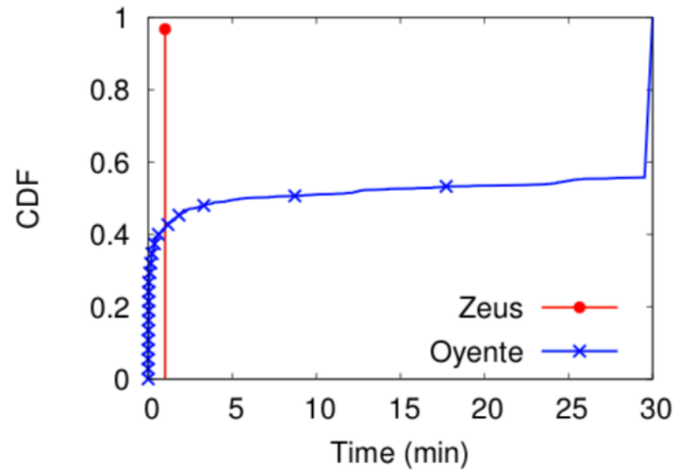
ZEUS

- **Pros**

Fully Automated

  - Fast + scalable

- **Cons**
  - Only works on Solidity code (not on bytecode)
  - Only works for pre-defined properties
  - Does not give soundness guarantees (transformations are not semantics preserving + security invariants are not proven sound)
  - Based (as Oyente) on flawed semantics



(d) Verification time in minutes.

ZEUS

- **Pros**

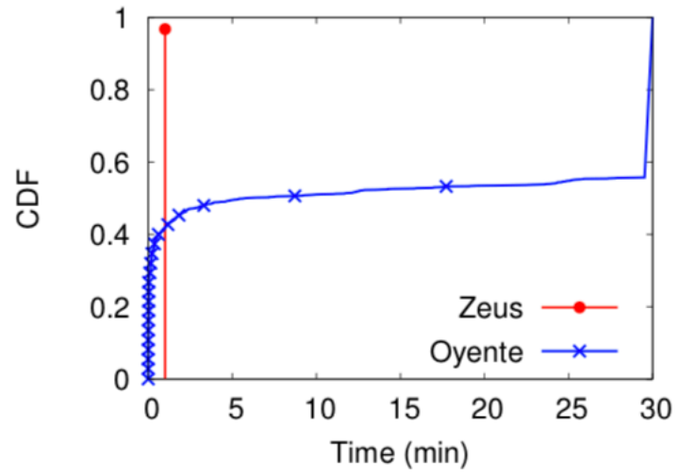
  Fully Automated

  - Fast + scalable

- **Cons**

  - Only works on Solidity code (not on bytecode)
  - Only works for pre-defined properties
  - Does not give soundness guarantees (transformations are not semantics preserving + security invariants are not proven sound)
  - Based (as Oyente) on flawed semantics

  No soundness guarantees



(d) Verification time in minutes.

EtherTrust

https://www.netidee.at/ethertrust

- First provably sound static analyzer for Ethereum smart contracts (i.e., it returns security guarantees)
  - previous ones focus on bug finding
- Outperforms the competitors in precision and performance
- Reachability analysis: suffices to check various interesting security properties

Specific application domain abstractions: T for unknown values, α for address of running contract, abstract memory representation, and most notably *calls to unknown contracts*

Static analysis for Ethereum smart contracts

- Approach: abstract the EVM small-step semantics into Horn clauses that can be analysed using Z3

$$c^* \qquad \Gamma, s_{c^*} :: S \qquad \rightarrow^* \qquad S' + +S$$

$$\Delta \quad \cup \quad \Pi_s \quad \vdash \quad \Pi \quad :> \quad \Pi_{S'}$$

Static analysis for Ethereum smart contracts

- Approach: abstract the EVM small-step semantics into Horn clauses that can be analysed using Z3

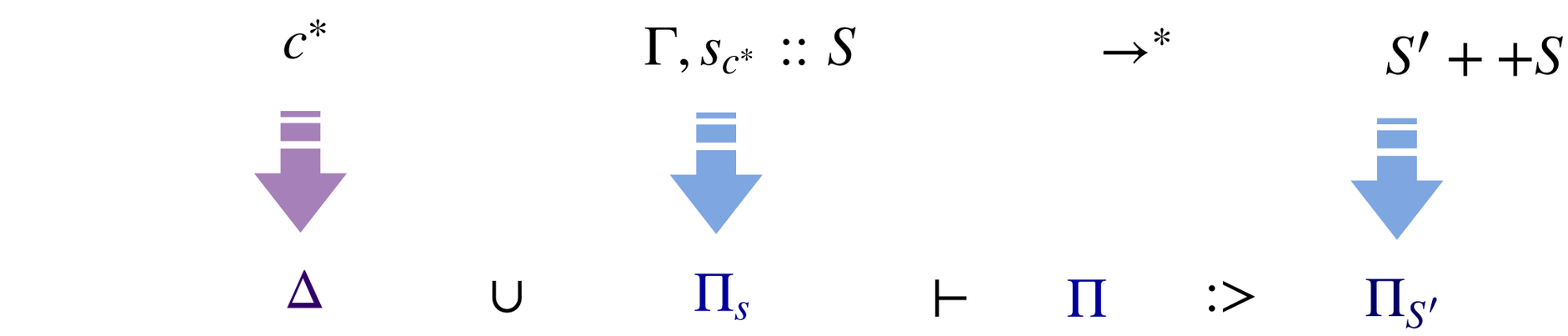
$$c^* \qquad \Gamma, s_{c^*} :: S \qquad \rightarrow^* \qquad S' + + S$$

$$\Delta \quad \cup \quad \Pi_s \quad \vdash \quad \Pi \quad :> \quad \Pi_{S'}$$

set of horn clauses over approximating the semantics of the code of contract c*

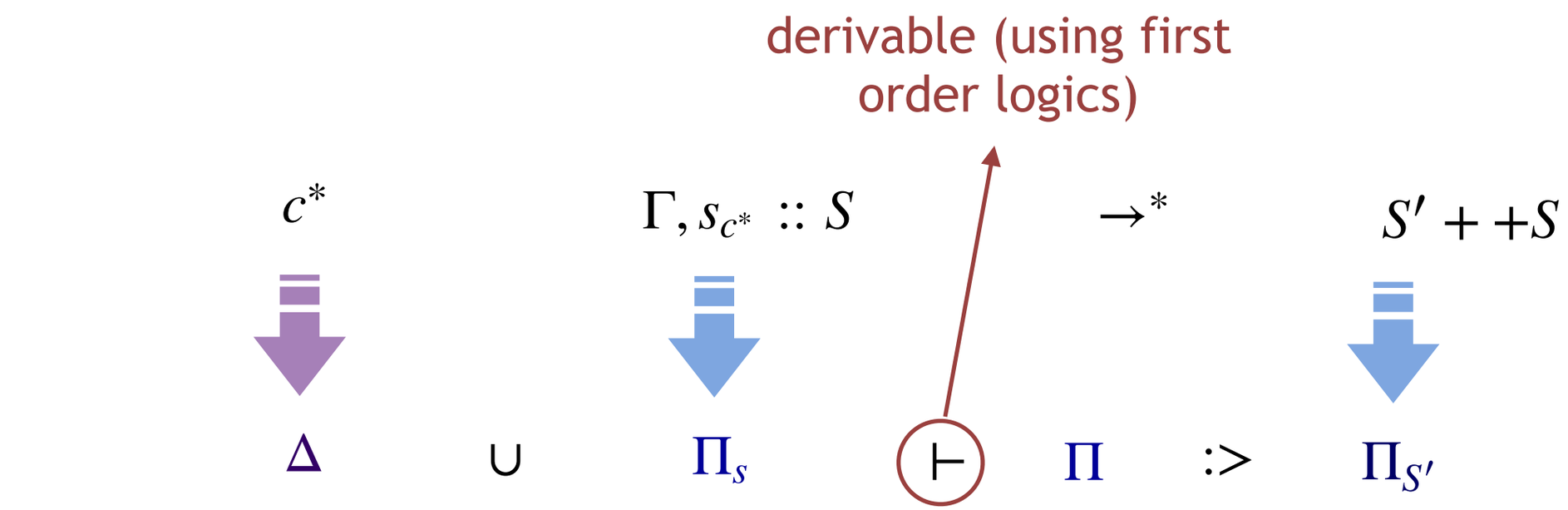
Static analysis for Ethereum smart contracts

- Approach: abstract the EVM small-step semantics into Horn clauses that can be analysed using Z3



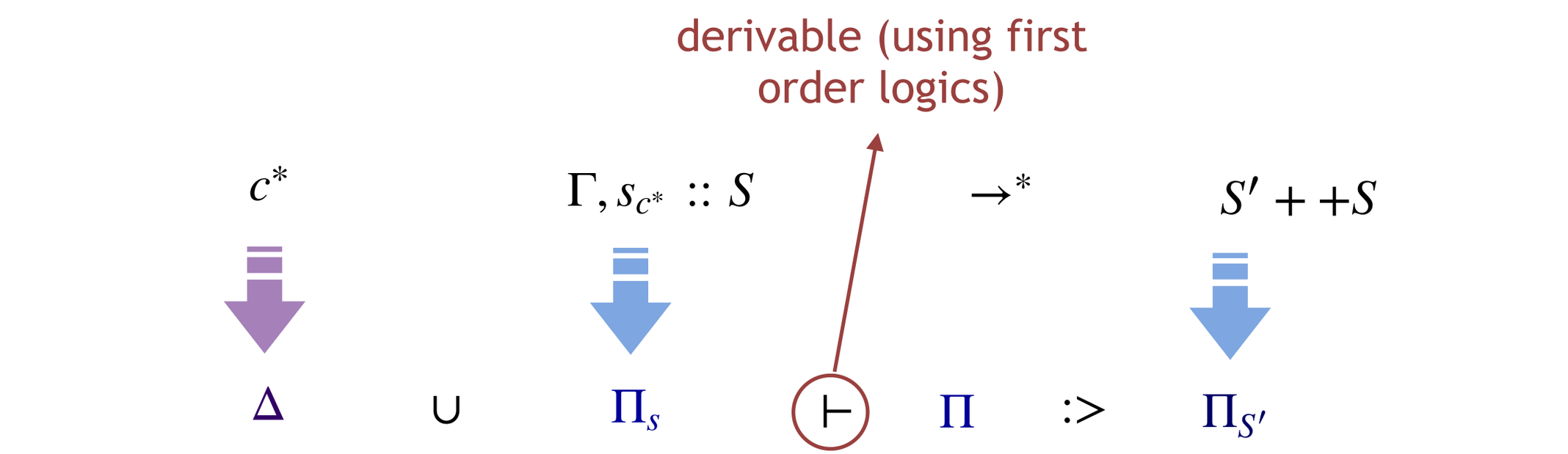
Static analysis for Ethereum smart contracts

- Approach: abstract the EVM small-step semantics into Horn clauses that can be analysed using Z3



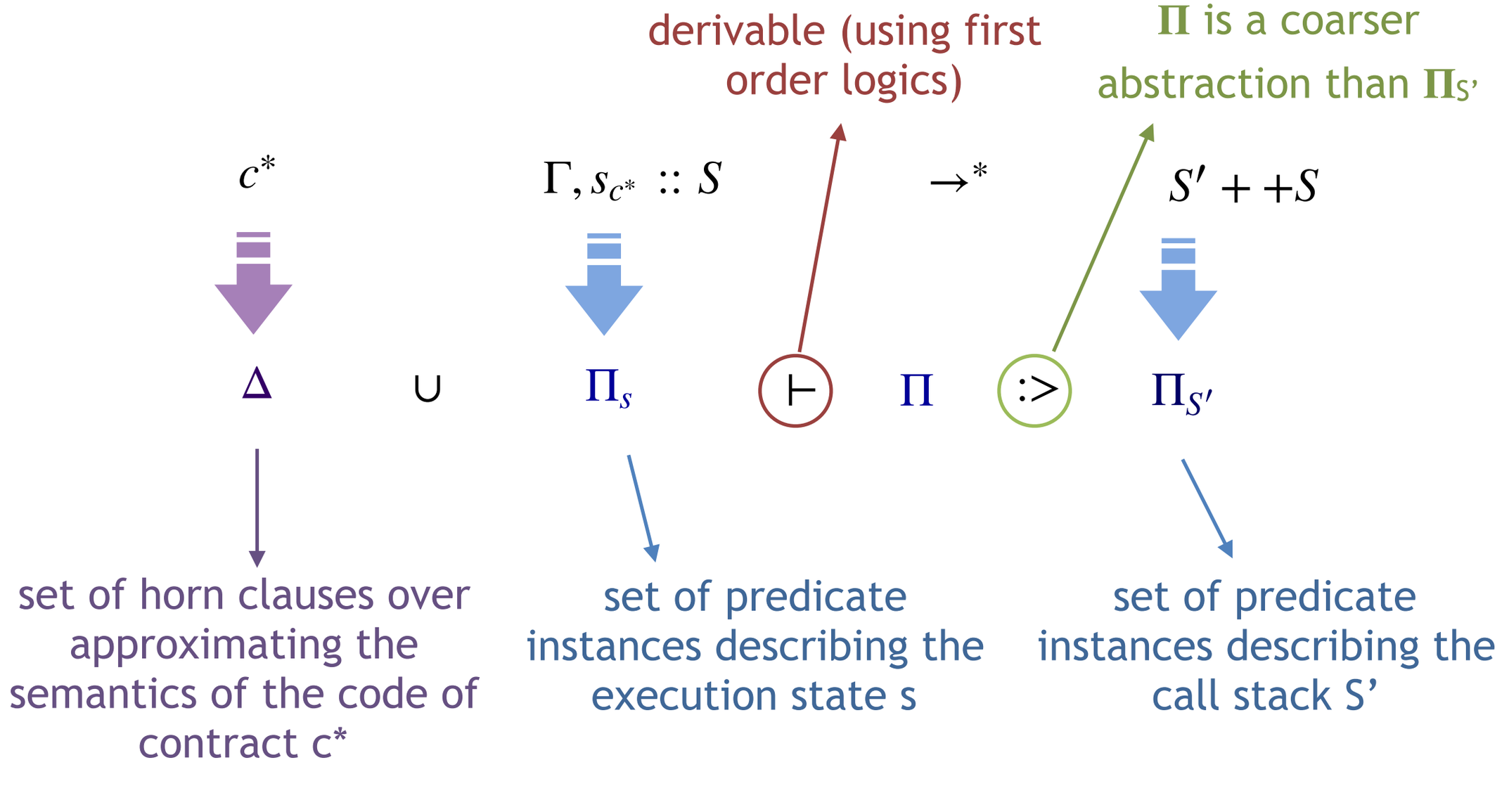
Static analysis for Ethereum smart contracts

- Approach: abstract the EVM small-step semantics into Horn clauses that can be analysed using Z3

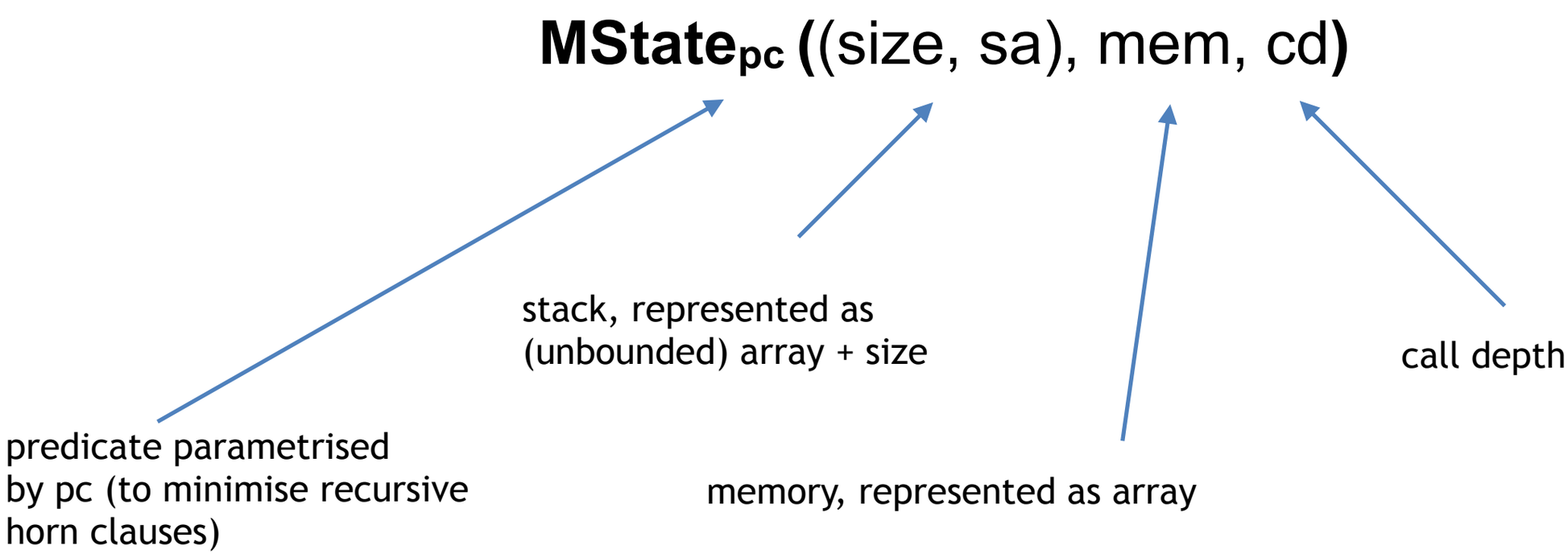


Static analysis for Ethereum smart contracts

- Approach: abstract the EVM small-step semantics into Horn clauses that can be analysed using Z3



State abstraction



- Similar predicates for
  - global state
  - execution environment

Horn clause encoding

- Execution steps modelled as Horn clauses
  - Horn clauses are generated according to the opcodes located at each pc
  - Example: Machine state rule for pc with opcode **ADD**

Horn clause encoding

- Execution steps modelled as Horn clauses
  - Horn clauses are generated according to the opcodes located at each pc
  - Example: Machine state rule for pc with opcode **ADD**

$$\mathbf{MState_{pc}(}(\text{size, sa}), \text{aw, cd}\mathbf{)}$$
$$\wedge\ \text{size} >= 2$$
$$\wedge\ x = \text{sa[size-1]}$$
$$\wedge\ y = \text{sa[size-2]}$$
$$\Rightarrow \mathbf{MState_{pc+1}(}(\text{size-1, sa[size-2} \rightarrow x+y]), \text{mem, cd}\mathbf{)}$$

Horn clause encoding

- Execution steps modelled as Horn clauses
  - Horn clauses are generated according to the opcodes located at each pc
  - Example: Machine state rule for pc with opcode **ADD**

$$\mathbf{MState_{pc}}((\text{size}, \text{sa}), \text{aw}, \text{cd})$$
$$\wedge\ \text{size} >= 2 \quad \leftarrow \text{simple range check}$$
$$\wedge\ x = \text{sa}[\text{size}-1]$$
$$\wedge\ y = \text{sa}[\text{size}-2]$$
$$\Rightarrow \mathbf{MState_{pc+1}}((\text{size}-1, \text{sa}[\text{size}-2 \rightarrow x+y]), \text{mem}, \text{cd})$$

state predicate for next pc is implied

stack is updated

Horn clause encoding

- Execution steps modelled as Horn clauses
  - Horn clauses are generated according to the opcodes located at each pc
  - Example: Machine state rule for pc with opcode **ADD**

MState$_{pc}$((size, sa), aw, cd**)**

∧ size >= 2 ← simple range check

∧ x = sa[size-1]

∧ y = sa[size-2]

⇒ **MState$_{pc+1}$(**(size-1, sa[size-2 → x+y]), mem, cd**)**

MState$_{pc}$((size, sa), mem, cd**)**

⇒ **Exc(**cd**)**

state predicate for next pc is implied

stack is updated

Horn clause encoding

- Execution steps modelled as Horn clauses
  - Horn clauses are generated according to the opcodes located at each pc
  - Example: Machine state rule for pc with opcode **ADD**

$$\mathbf{MState_{pc}}((size, sa), aw, cd)$$
$$\wedge\ size >= 2$$
$$\wedge\ x = sa[size-1]$$
$$\wedge\ y = sa[size-2]$$
$$\Rightarrow \mathbf{MState_{pc+1}}((size-1, sa[size-2 \rightarrow x+y]), mem, cd)$$

simple range check

state predicate for next pc is implied

stack is updated

$$\mathbf{MState_{pc}}((size, sa), mem, cd)$$
$$\Rightarrow \mathbf{Exc}(cd)$$

as we do not track gas explicitly, we assume the execution to stop in every step as it runs out of gas

Horn clause encoding

- Execution steps modelled as Horn clauses
  - Horn clauses are generated according to the opcodes located at each pc
  - Example: Machine state rule for p… …ode **ADD**

MState$_{pc}$((size, sa), aw, cd**)** … sa), mem, cd**)**

∧ size >= 2 ← simple…

∧ x = sa[size-1]

∧ y = sa[size-2]

⇒ **MState$_{pc+1}$**((size-1, s… …]), mem, cd**)**

as we do not track gas explicitly, we assume the execution to stop in every step as it runs out of gas

state predicate for next pc is implied

stack is updated

It's a bit more complicated than that

Abstract Domain

$$\mathsf{MState}_{pc}\ ((size, \boxed{sa}), ma, cd)$$
$$\in \mathbb{N} \rightarrow \hat{D}$$

Abstract domain $\hat{D} = \mathbb{Z} \cup \{\alpha\} \cup \{\top\}$

- $\mathbb{Z}$: represents concretely known values
- α: represents the address of the account under analysis
- $\top$: represents all potential values

Abstract Domain

$$\mathsf{MState}_{pc}\ ((size, \boxed{sa}), ma, cd)$$

$$\in \mathbb{N} \rightarrow \hat{D}$$

Common technique from abstract interpretation

Abstract domain $\hat{D} = \mathbb{Z} \cup \{\alpha\} \cup \{\top\}$

represents concretely known values

represents the address of the account under analysis

represents all potential values

Why using an abstract domain?

Why using an abstract domain?

- There are a lot of values that we do not know statically:

$$\text{TIMESTAMP}, \text{PUSH}\,2, \text{ADD}\,\textbf{?}$$

Why using an abstract domain?

- There are a lot of values that we do not know statically:

only known when contract is called on blockchain

TIMESTAMP, PUSH 2, ADD ?

Why using an abstract domain?

- There are a lot of values that we do not know statically:

only known when contract is called on blockchain

TIMESTAMP, PUSH 2, ADD ?

Why using an abstract domain?

- There are a lot of values that we do not know statically:

only known when contract is called on blockchain

TIMESTAMP, PUSH 2, ADD ?

- Sometimes we still want to be precise:

ADDRESS, BALANCE

Why using an abstract domain?

- There are a lot of values that we do not know statically:

only known when contract is called on blockchain

TIMESTAMP, PUSH 2, ADD ?

- Sometimes we still want to be precise:

only known after contract is published on blockchain

ADDRESS, BALANCE

takes address as argument and returns the balance of the corresponding account

Why using an abstract domain?

- There are a lot of values that we do not know statically:

only known when contract is called on blockchain

TIMESTAMP, PUSH 2, ADD ?

- Sometimes we still want to be precise:

only known after contract is published on blockchain

ADDRESS, BALANCE

takes address as argument and returns the balance of the corresponding account

always results in pushing the balance of the executing account

Why using an abstract domain?

- There are a lot of values that we do not know statically:

only known when contract is called on blockchain

TIMESTAMP, PUSH 2, ADD ?

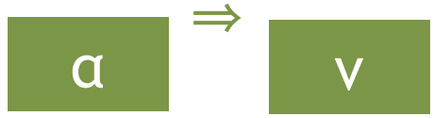
- Sometimes we still want to be precise:

only known after contract is published on blockchain

ADDRESS, BALANCE

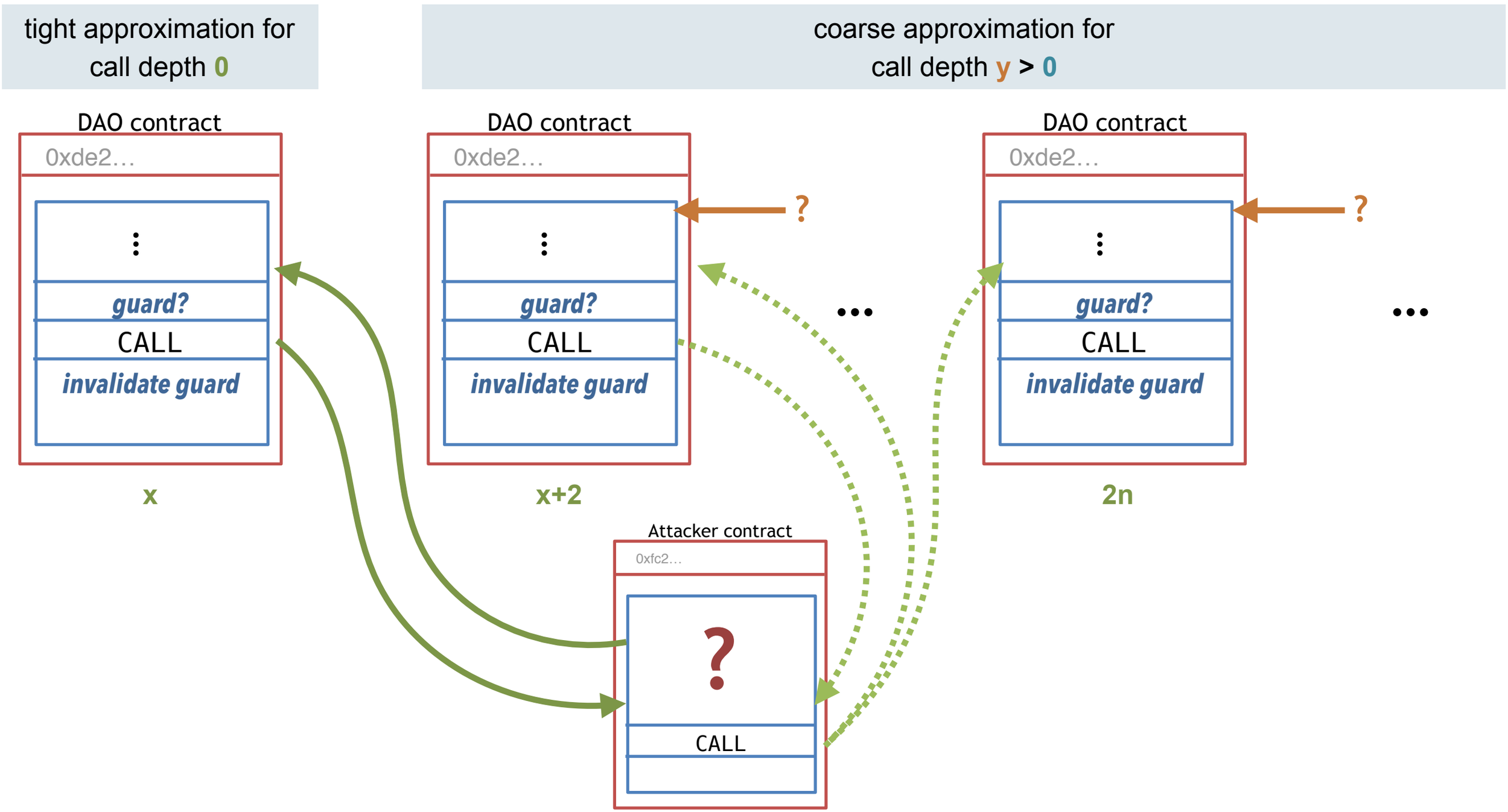
takes address as argument and returns the balance of the corresponding account

always results in pushing the balance of the executing account

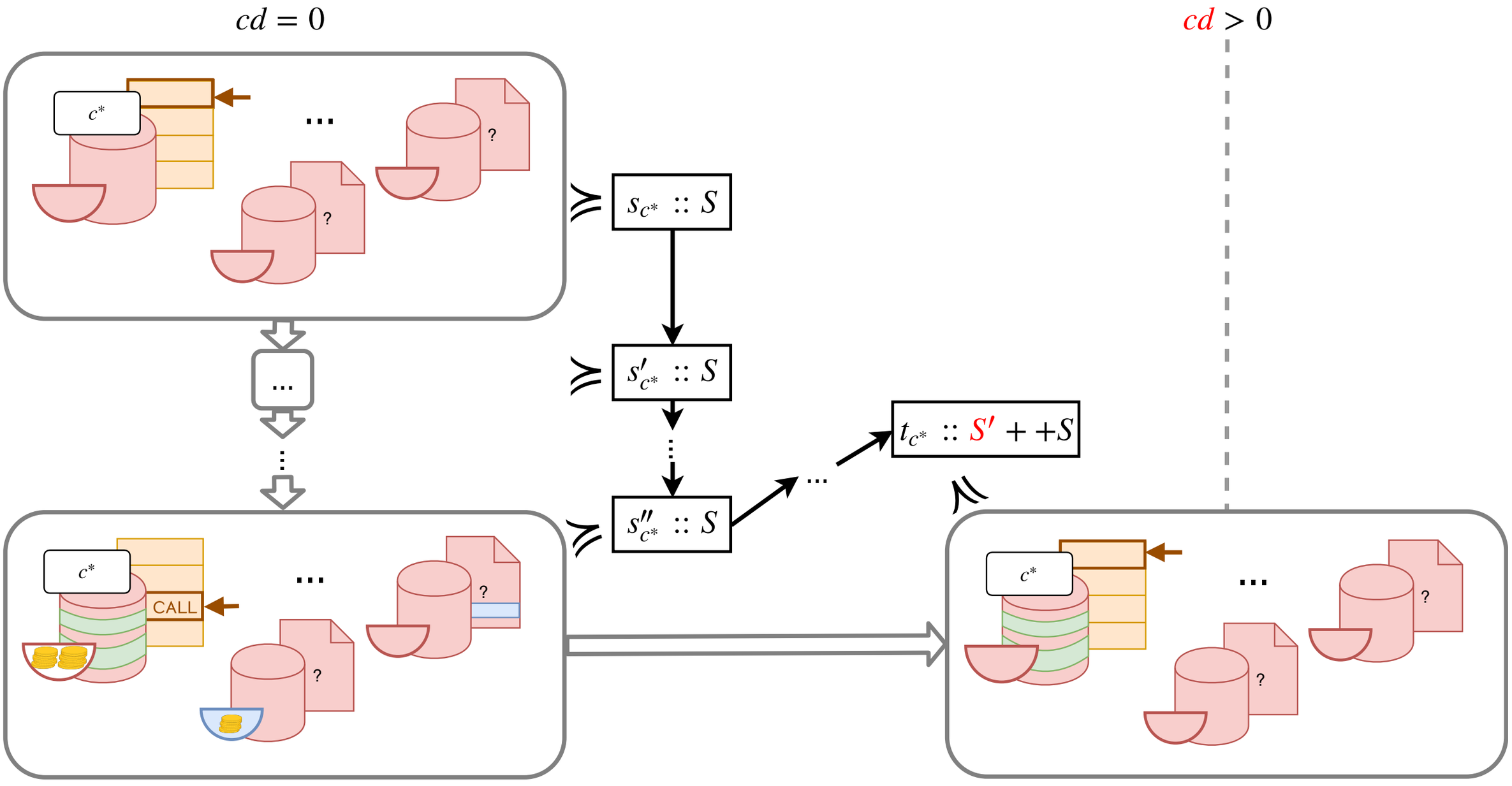


Abstracting calls - Intuition

- For analysing a specific contract all its executions need to be approximated



Abstract semantics - Call rule



Abstract semantics - Call rule

preconditions are checked: enough elements on the stack, enough gas

$$\mathsf{MState}_{pc}\,((size, sa), ma, cd) \wedge size > 3 \wedge \hat{va} = sa[size-2] \wedge \hat{va} \mathrel{\widehat{\leq}} \hat{b}$$

$$\wedge \mathsf{GState}_{pc}\,(\alpha, \hat{b}, sta, cd) \wedge \boxed{cd' > cd} \Rightarrow \mathsf{MState}_{\boxed{0}}\,(\boxed{(0, \lambda x.\top)}, \boxed{\lambda x.0}, cd')$$

contract might be reentered at an arbitrarily higher call depth

execution starts at pc 0 in fresh machine state: empty stack + memory initialised to all zeros

$$\mathsf{MState}_{pc}\,((size, sa), ma, cd) \wedge size > 3 \wedge \hat{va} = sa[size-2] \wedge va \mathrel{\widehat{\leq}} \hat{b}$$

$$\wedge \mathsf{GState}_{pc}\,(\alpha, \hat{b}, sta, cd) \wedge cd' > cd \Rightarrow \mathsf{ExEnv}\,(\boxed{\alpha}, \boxed{\top}, cd')$$

when reentering the active account at the point of calling is (still) the actor

the input to the reentering call is unknown

Abstract semantics - Call rules

$$\mathsf{MState}_{pc}\left((size, sa), ma, cd\right) \wedge size > 3 \wedge \hat{va} = sa[size-2] \wedge \hat{va} \mathrel{\widehat{\leq}} \hat{b}$$
$$\wedge \mathsf{GState}_{pc}\left(\alpha, \hat{b}, sta, cd\right) \wedge cd' > cd \Rightarrow \mathsf{GState}_{0}\left(\alpha, \top, sta, cd'\right)$$

the global storage of the active account is preserved

the balance might be arbitrarily changed

$$\mathsf{MState}_{pc}\left((size, sa), ma, cd\right) \wedge size > 3 \wedge \hat{va} = sa[size-2] \wedge \mathsf{GState}_{pc}\left(\alpha, \hat{b}, sta, cd\right)$$
$$\wedge \hat{va} \mathrel{\widehat{\leq}} \hat{b} \wedge \mathsf{GState}_{pc}\left(\dot{a}, \hat{b}^{*}, sta^{*}, cd\right) \wedge \dot{a} \neq \alpha \wedge cd' > cd \Rightarrow \mathsf{GState}_{0}\left(\dot{a}, \top, [\top], cd'\right)$$

all addresses different from the actor can have arbitrary storage (all positions mapped to T)

This is actually an artefact: the storage of other accounts but the active account cannot be accessed anyways given that only plain calls are executed

Checking for single-entrancy

- How to check for single-entrancy now?
- Simple example:

```
contract Bob {
 bool sent = false;

 function ping(address c) {
  if (!sent){
    c.call.value(2)();
    sent = true; }
 }
}
```

Checking for single-entrancy

- How to check for single-entrancy now?
- Simple example:

```
contract Bob {
 bool sent = false;

 function ping(address c) {
  if (!sent){
    c.call.value(2)();
    sent = true; }
 }
}
```

Transfers 2 wei to the address given as argument

Checking for single-entrancy

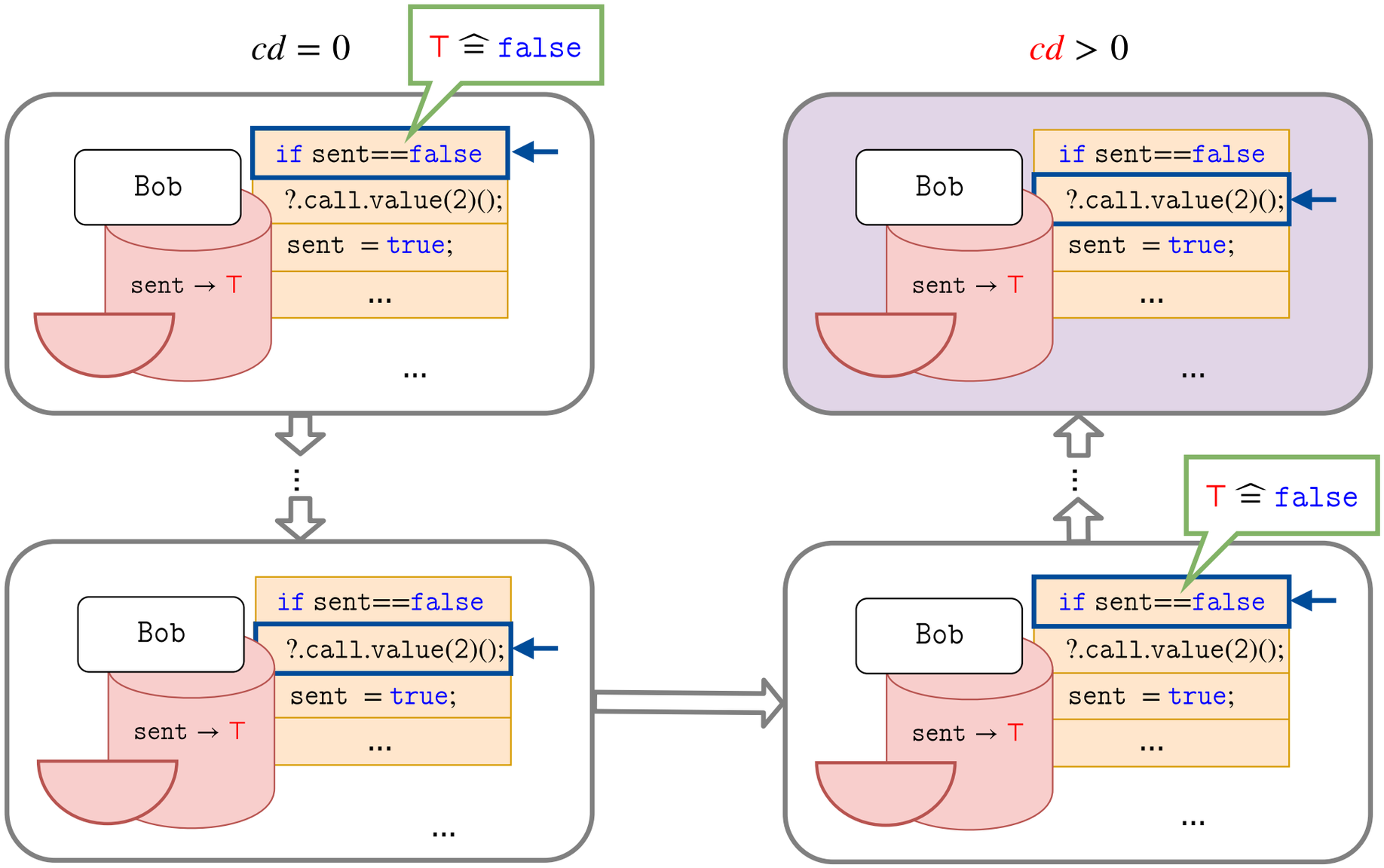
- How to check for single-entrancy now?
- Simple example:

```
contract Bob {
 bool sent = false;

 function ping(address c) {
  if (!sent){
   c.call.value(2)();
   sent = true; }
 }
}
```

Transfers 2 wei to the address given as argument

Can it ever happen that we execute this call while re-entering?

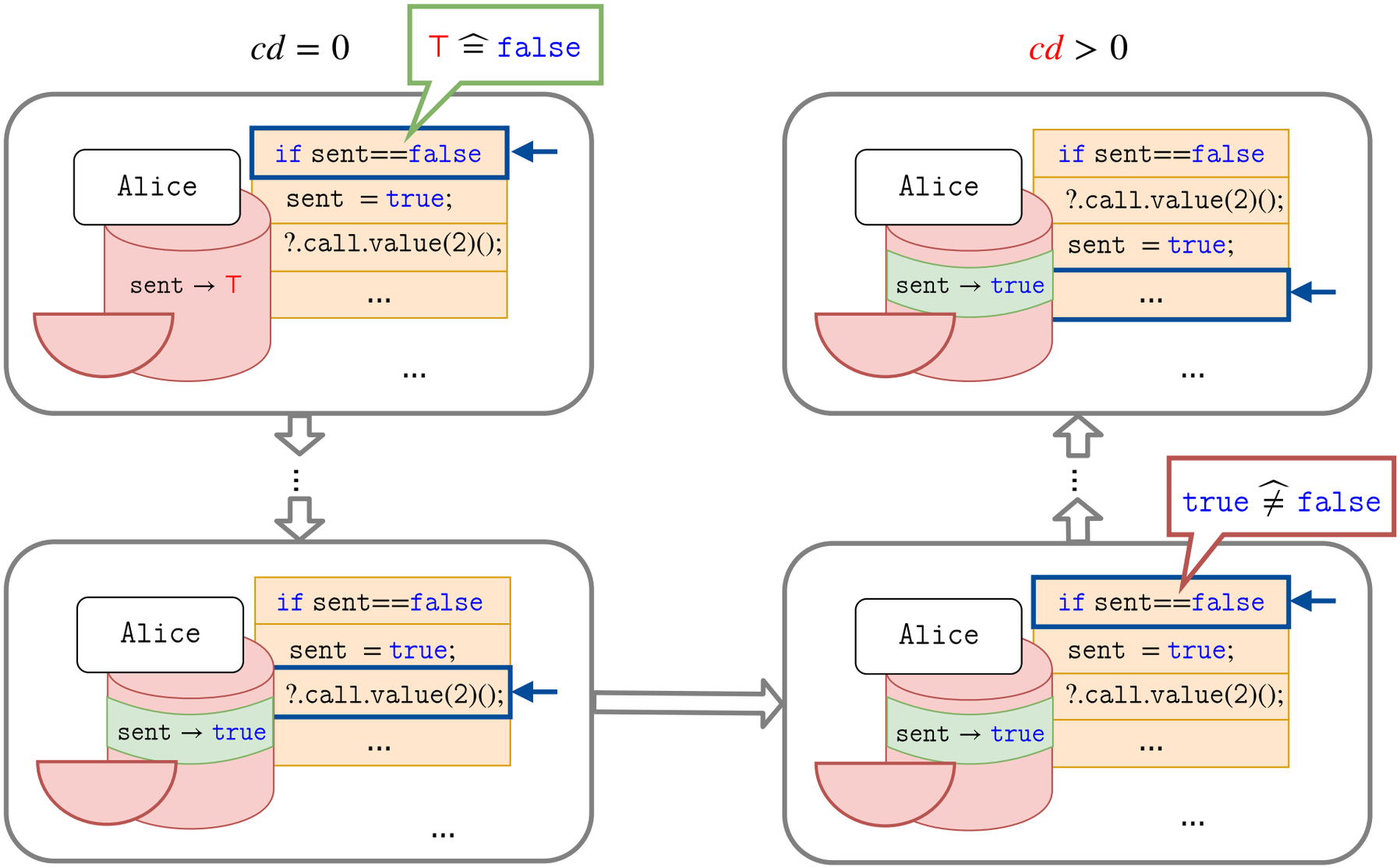
Detecting reentrancy



Reachability query

$cd > 0 \quad \wedge$ `?.call.value(2)();`

Proving single-entrancy



Reachability query

$cd > 0 \quad \wedge$ `?.call.value(2)();`

EtherTrust

- Approach scales to full EVM bytecode!
- We implemented **EtherTrust** - a tool for static analysing Ethereum bytecode

EtherTrust

- Approach scales to full EVM bytecode!
- We implemented **EtherTrust** - a tool for static analysing Ethereum bytecode

Average running time: much faster than the best state-of-the art bug finding tool… and **sound!**

10k	$\sum$	# ter. SE	# $\overline{\text{SE}}$	# ter. MI	# $\overline{\text{MI}}$	$\varnothing t$
O	148	18	12	(18)	3	26,5
ET		100	4	107	2	2,8

SE: Single entrancy
MI: Independence of miner controlled state
#X: Number of contracts reported to violate X
#ter. X: Number of contracts for which the analysis terminates

O: Oyente (state-of-the-art) bug finder
ET: EtherTrust

Simplifications in this tutorial

- Simplified gas treatment (constant gas cost of 1)
- Inherent exception propagation (all available gas is given to the caller)
- Simplified memory treatment (only memory cells are accessed, never fragments; word indexed memory)
- computations on logical (instead of bounded) integers
- No limits on call stack and machine stack
- Some interesting opcodes are omitted (DELEGATECALL, CALLCODE, **CREATE**, …)

Simplifications in this tutorial

- Simplified gas treatment (constant gas cost of 1)
- Inherent exception propagation (all available gas is given to the caller)
- Simplified memory treatment (only memory cells are accessed, never fragments; word indexed memory)
- computations on logical (instead of bounded) integers
- No limits on call stack and machine stack
- Some interesting opcodes are omitted (DELEGATECALL, CALLCODE, **CREATE**, …)

We are hiring PhDs and Postdocs!

logics LOGICAL METHODS IN COMPUTER SCIENCE

CALL FOR DOCTORAL STUDENTS

LOGICAL METHODS IN COMPUTER SCIENCE

DOCTORAL PROGRAM

TU Wien, TU Graz, and JKU Linz are seeking exceptionally talented and motivated students for their joint doctoral program LogiCS. The LogiCS doctoral program focuses on interdisciplinary research topics covering

computational logic | databases and artificial intelligence | computer-aided verification

security and privacy | cyber-physical systems | distributed systems

THE PROGRAM

LogiCS is a doctoral program focusing on logic and its applications in computer science. Successful applicants will work with and be supervised by leading researchers in the fields of: computational logic, databases and knowledge representation, computer-aided verification, security and privacy, cyber-physical systems, and distributed systems.

FACULTY MEMBERS

E. Bartocci | A. Biere | R. Bloem | A. Ciabattoni
G. Gottlob | T. Eiter | R. Grosu | L. Kovacs
M. Maffei | M. Ortiz | U. Schmid | M. Seidl
S. Szeider | G. Weissenbacher | S. Woltran

LOGIC IN AUSTRIA

The cities of Vienna, Graz, and Linz provide an exceptionally high quality of life and thriving logic in computer science community.

http://vsl2014.at • www.vcla.at
https://kgs.logic.at • http://arise.or.at

FINANCIAL SUPPORT

We are looking for doctoral students, where 30% of the positions are reserved for highly qualified female candidates. The doctoral positions are financed by 4 year scholarships according to the funding scheme of the Austrian Science Fund.

HOW TO APPLY

Detailed information about the application process is available on the LogiCS web-page

http://logic-cs.at/phd

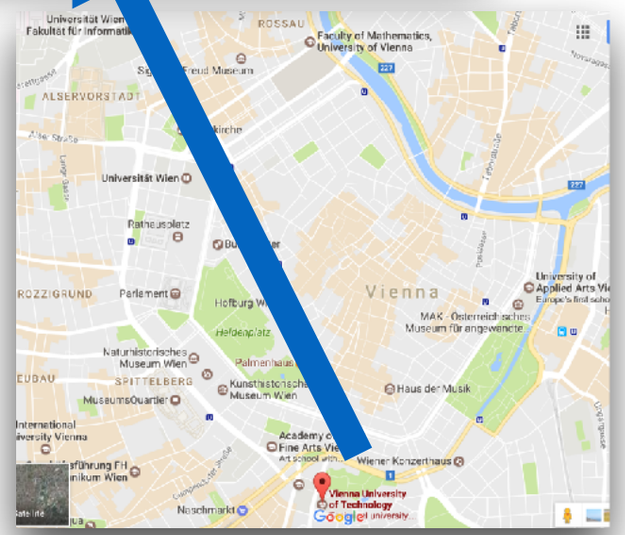
The applicants are expected to have completed an excellent diploma or master's degree in computer science, mathematics, or a related field. Applications by the candidates can be submitted on continuous basis till October 2018 until all positions are filled.

erc European Research Council

FWF Der Wissenschaftsfonds.

FFG Österreichische Forschungsförderungsgesellschaft

TU WIEN



Handling abstract values

- Abstract operations:

- Abstract comparisons:

Handling abstract values

- Abstract operations:

$$n \mathbin{\widehat{+}} m := n + m$$
$$\hat{v} \mathbin{\widehat{+}} \top := \top$$
$$\top \mathbin{\widehat{+}} \hat{v} := \top$$
$$\hat{v} \mathbin{\widehat{+}} \alpha := \top$$
$$\alpha \mathbin{\widehat{+}} \hat{v} := \top$$

- Abstract comparisons:

Handling abstract values

- Abstract operations:

$$n \mathbin{\widehat{+}} m := n + m$$

$$\left.\begin{array}{l} \hat{v} \mathbin{\widehat{+}} \top := \top \\ \top \mathbin{\widehat{+}} \hat{v} := \top \end{array}\right\}$$

No constraints are collected for computations with T

$$\left.\begin{array}{l} \hat{v} \mathbin{\widehat{+}} \alpha := \top \\ \alpha \mathbin{\widehat{+}} \hat{v} := \top \end{array}\right\}$$

Abstract address is ‘supertyped’ once it is modified

- Abstract comparisons:

Handling abstract values

- Abstract operations:

$$n \mathbin{\widehat{+}} m := n + m$$

$$\left.\begin{array}{l} \hat{v} \mathbin{\widehat{+}} \top := \top \\ \top \mathbin{\widehat{+}} \hat{v} := \top \end{array}\right\}$$

No constraints are collected for computations with T

$$\left.\begin{array}{l} \hat{v} \mathbin{\widehat{+}} \alpha := \top \\ \alpha \mathbin{\widehat{+}} \hat{v} := \top \end{array}\right\}$$

Abstract address is ‘supertyped’ once it is modified

- Abstract comparisons:

$$n \mathbin{\widehat{=}} m := \{n = m\}$$
$$\alpha \mathbin{\widehat{=}} \alpha := \{true\}$$
$$\hat{v} \mathbin{\widehat{=}} \top := \{\mathsf{true}, \mathsf{false}\}$$
$$\top \mathbin{\widehat{=}} \hat{v} := \{\mathsf{true}, \mathsf{false}\}$$
$$\hat{v} \mathbin{\widehat{=}} \alpha := \{\mathsf{true}, \mathsf{false}\}$$
$$\alpha \mathbin{\widehat{=}} \hat{v} := \{\mathsf{true}, \mathsf{false}\}$$

Handling abstract values

- Abstract operations:

$$n \mathbin{\widehat{+}} m := n + m$$

$$\left.\begin{array}{l} \hat{v} \mathbin{\widehat{+}} \top := \top \\ \top \mathbin{\widehat{+}} \hat{v} := \top \end{array}\right\}$$

No constraints are collected for computations with T

$$\left.\begin{array}{l} \hat{v} \mathbin{\widehat{+}} \alpha := \top \\ \alpha \mathbin{\widehat{+}} \hat{v} := \top \end{array}\right\}$$

Abstract address is ‘supertyped’ once it is modified

- Abstract comparisons:

$$n \mathbin{\widehat{=}} m := \{n = m\}$$

$$\alpha \mathbin{\widehat{=}} \alpha := \{true\}$$

$$\left.\begin{array}{l} \hat{v} \mathbin{\widehat{=}} \top := \{\text{true, false}\} \\ \top \mathbin{\widehat{=}} \hat{v} := \{\text{true, false}\} \\ \hat{v} \mathbin{\widehat{=}} \alpha := \{\text{true, false}\} \\ \alpha \mathbin{\widehat{=}} \hat{v} := \{\text{true, false}\} \end{array}\right\}$$

Comparisons with unknown values evaluate to both true and false

Abstract ADD - revisited

$\mathsf{MState}_{pc}\,((size, sa), ma, cd)$
$\wedge\, size > 1$
$\wedge\, \hat{x} = sa[size - 1]$
$\wedge\, \hat{y} = sa[size - 2]$
$\Rightarrow \mathsf{MState}_{pc+1}\,((size - 1, sa[size - 2 \rightarrow \hat{x} \mathbin{\widehat{+}} \hat{y}]), ma, cd)$

$\mathsf{MState}_{pc}\,((size, sa), ma, cd)$
$\Rightarrow \mathsf{Exc}\,(cd)$

abstract operations are used!

Abstract Memory access

$$ma \in \hat{D}/\{\alpha\} \rightarrow \hat{D}$$

Memory positions can't be α

Memory values come from the abstract domain

Abstract Memory access

$$ma \in \hat{D}/\{\alpha\} \rightarrow \hat{D}$$

Memory positions can't be α

Memory values come from the abstract domain

MSTORE

$$\begin{aligned}
&\mathsf{MState}_{pc}\ ((size, sa), ma, cd) \\
&\wedge\ size > 1 \\
&\wedge\ \hat{x} = sa[size - 1] \\
&\wedge\ \hat{p} = (\hat{x} = \alpha)\ ?\ \top\ :\ \hat{x} \\
&\wedge\ \hat{y} = sa[size - 2] \\
&\Rightarrow \mathsf{MState}_{pc+1}\ ((size - 2, sa), ma[\hat{p} \rightarrow \hat{y}], cd)
\end{aligned}$$

Abstract Memory access

$$ma \in \hat{D}/\{\alpha\} \rightarrow \hat{D}$$

Memory positions can't be α

Memory values come from the abstract domain

MSTORE

$$\begin{aligned}
&\mathsf{MState}_{pc}\ ((size, sa), ma, cd) \\
&\wedge\ size > 1 \\
&\wedge\ \hat{x} = sa[size - 1] \\
&\wedge\ \hat{p} = (\hat{x} = \alpha)\ ?\ \top\ :\ \hat{x} \\
&\wedge\ \hat{y} = sa[size - 2] \\
&\Rightarrow \mathsf{MState}_{pc+1}\ ((size - 2, sa), ma[\hat{p} \rightarrow \hat{y}], cd)
\end{aligned}$$

Instead of writing to α, we write to T

writing to T means "writing everywhere"

Abstract Memory access

$$ma \in \hat{D}/\{\alpha\} \rightarrow \hat{D}$$

Memory positions can't be α

Memory values come from the abstract domain

MSTORE

$\mathsf{MState}_{pc}\ ((size, sa), ma, cd)$
$\wedge\ size > 1$
$\wedge\ \hat{x} = sa[size - 1]$
$\wedge\ \hat{p} = (\hat{x} = \alpha)\ ?\ \top\ :\ \hat{x}$
$\wedge\ \hat{y} = sa[size - 2]$
$\Rightarrow \mathsf{MState}_{pc+1}\ ((size - 2, sa), ma[\hat{p} \rightarrow \hat{y}], cd)$

Instead of writing to α, we write to T

MLOAD

$\mathsf{MState}_{pc}\ ((size, sa), ma, cd)$
$\wedge\ size > 0$
$\wedge\ sa[size - 1] \in \{\alpha, \top\}$
$\Rightarrow \mathsf{MState}_{pc+1}\ ((size - 2, sa[size - 1 \rightarrow \top]), ma, cd)$

writing to T means "writing everywhere"

Abstract Memory access

$$ma \in \hat{D}/\{\alpha\} \rightarrow \hat{D}$$

Memory positions can't be α

Memory values come from the abstract domain

MSTORE

$\mathsf{MState}_{pc}\ ((size, sa), ma, cd)$
$\wedge\ size > 1$
$\wedge\ \hat{x} = sa[size - 1]$
$\wedge\ \hat{p} = (\hat{x} = \alpha)\ ?\ \top\ :\ \hat{x}$
$\wedge\ \hat{y} = sa[size - 2]$
$\Rightarrow \mathsf{MState}_{pc+1}\ ((size - 2, sa), ma[\hat{p} \rightarrow \hat{y}], cd)$

Instead of writing to α, we write to T

writing to T means "writing everywhere"

MLOAD

$\mathsf{MState}_{pc}\ ((size, sa), ma, cd)$
$\wedge\ size > 0$
$\wedge\ sa[size - 1] \in \{\alpha, \top\}$
$\Rightarrow \mathsf{MState}_{pc+1}\ ((size - 2, sa[size - 1 \rightarrow \top]), ma, cd)$

Instead of "reading from everywhere", we simply read T

Abstract Memory access - continued

$$\mathsf{MState}_{pc}\ ((size, sa), ma, cd)$$
$$\wedge\ size > 0$$
$$\wedge\ n = sa[size - 1]$$
$$\Rightarrow \mathsf{MState}_{pc+1}\ ((size, sa[size - 1 \rightarrow ma[n]]), ma, cd)$$

MLOAD

$$\mathsf{MState}_{pc}\ ((size, sa), ma, cd)$$
$$\wedge\ size > 0$$
$$\wedge\ n = sa[size - 1]$$
$$\Rightarrow \mathsf{MState}_{pc+1}\ ((size, sa[size - 1 \rightarrow ma[\top]]), ma, cd)$$

Abstract Memory access - continued

$$\mathsf{MState}_{pc}\ ((size, sa), ma, cd)$$
$$\wedge\ size > 0$$
$$\wedge\ n = sa[size - 1]$$
$$\Rightarrow \mathsf{MState}_{pc+1}\ ((size, sa[size - 1 \rightarrow ma[n]]), ma, cd)$$

MLOAD

When reading the memory at a concrete position, we additionally need to read from T, as there we can find the values that have been written everywhere

$$\mathsf{MState}_{pc}\ ((size, sa), ma, cd)$$
$$\wedge\ size > 0$$
$$\wedge\ n = sa[size - 1]$$
$$\Rightarrow \mathsf{MState}_{pc+1}\ ((size, sa[size - 1 \rightarrow ma[\top]]), ma, cd)$$