# A Concurrent Perspective on Smart Contracts

Ilya Sergey

UCL

Aquinas Hobor

NUS National University of Singapore

1st Workshop on Trusted Smart Contracts

7 April 2017

```
class ConcurrentQueue <E> {
   public synchronized void enqueue(E elem) {…}
   public synchronized E dequeue() {…}
}
```

```
class ConcurrentQueue <E> {
   public synchronized void enqueue(E elem) {…}
   public synchronized E dequeue() {…}
}
```

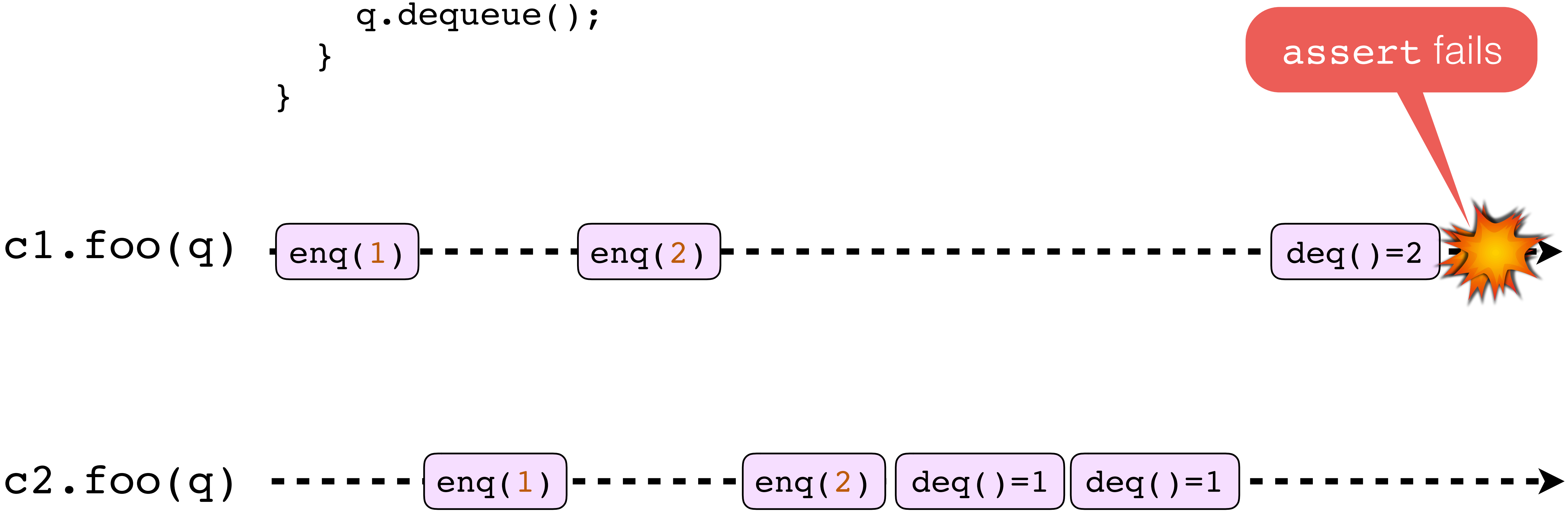
```
class MyQClient {
  public void foo (ConcurrentQueue<Integer> q) {
    …
    q.enqueue(1);
    q.enqueue(2);
    doStuff();
    Integer i = q.dequeue();
    assert (i == 1);
    q.dequeue();
  }
}
```

```java
class MyQClient {
  public void foo (ConcurrentQueue<Integer> q) {
    …
    q.enqueue(1);
    q.enqueue(2);
    doStuff();
    Integer i = q.dequeue();
    assert (i == 1);
    q.dequeue();
  }
}
```

```java
Queue q = new ConcurentQueue<Integer>();
MyQClient c1 = new MyQClient();
MyQClient c2 = new MyQClient();
```

```
c1.foo(q)   ||   c2.foo(q)
```

```
class MyQClient {
  public void foo (ConcurrentQueue<Integer> q) {
    …
    q.enqueue(1);
    q.enqueue(2);
    doStuff();
    Integer i = q.dequeue();
    assert (i == 1);
    q.dequeue();
  }
}
```



assert fails

c1.foo(q)

enq(1)

enq(2)

deq()=2

c2.foo(q)

enq(1)

enq(2)

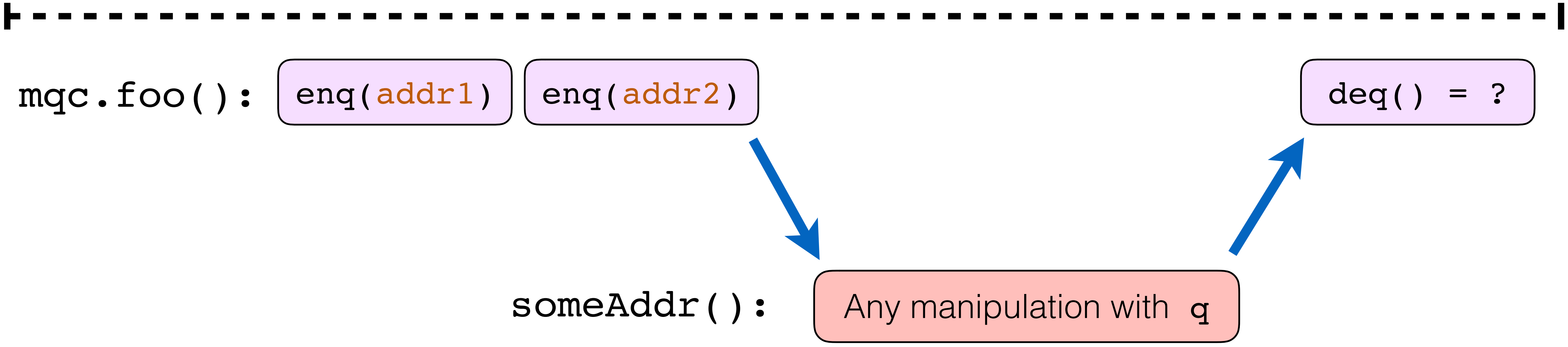
deq()=1

deq()=1

```
contract MyQContract {

  Queue q = QueueContract(0x1d11e5fbe221);

  function foo() {
    …
    q.enqueue(addr1);
    q.enqueue(addr2);
    someAddr.call.value(…);
    address i = q.dequeue();
    // Assuming i == addr1
    i.send(reward);
    q.dequeue();
  }
}
```

```
contract MyQContract {

  Queue q = QueueContract(0x1d11e5fbe221);

  function foo() {
    …
    q.enqueue(addr1);
    q.enqueue(addr2);
    someAddr.call.value(…);
    address i = q.dequeue();
    // Assuming i == addr1
    i.send(reward);
    q.dequeue();
  }
}
```

Transaction



mqc.foo():

enq(addr1)

enq(addr2)

deq() = ?

someAddr():

Any manipulation with q

*Accounts* using ***smart contracts*** in a blockchain
are like
*threads* using ***concurrent objects*** in shared memory.

*Accounts* using ***smart contracts*** in a blockchain
are like
*threads* using ***concurrent objects*** in shared memory.

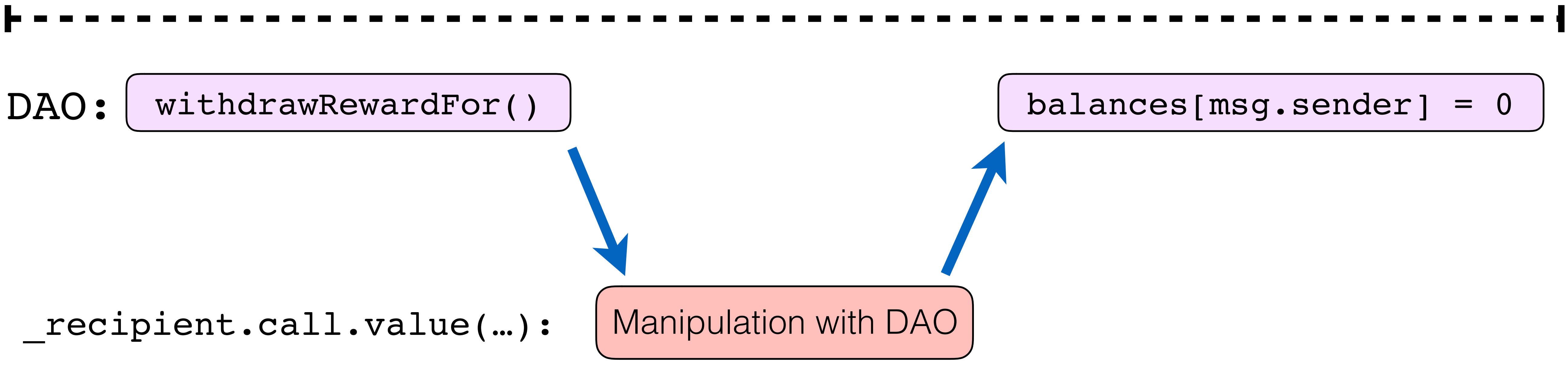
| | | |
|---|---|---|
| contract state | — | object state |
| call/send | — | context switching |
| Reentrancy | — | (Un)cooperative multitasking |

# Reentrancy and multitasking

```
 // Burn DAO Tokens
 Transfer(msg.sender, 0, balances[msg.sender]);
 withdrawRewardFor(msg.sender); // be nice, and get his rewards
 totalSupply -= balances[msg.sender];
 balances[msg.sender] = 0;
 paidOut[msg.sender] = 0;
 return true;
}
```

# Reentrancy and multitasking

```
 // Burn DAO Tokens
 Transfer(msg.sender, 0, balances[msg.sender]);
 withdrawRewardFor(msg.sender); // be nice, and get his rewards
 totalSupply -= balances[msg.sender];
 balances[msg.sender] = 0;
 paidOut[msg.sender] = 0;
 return true;
}
```



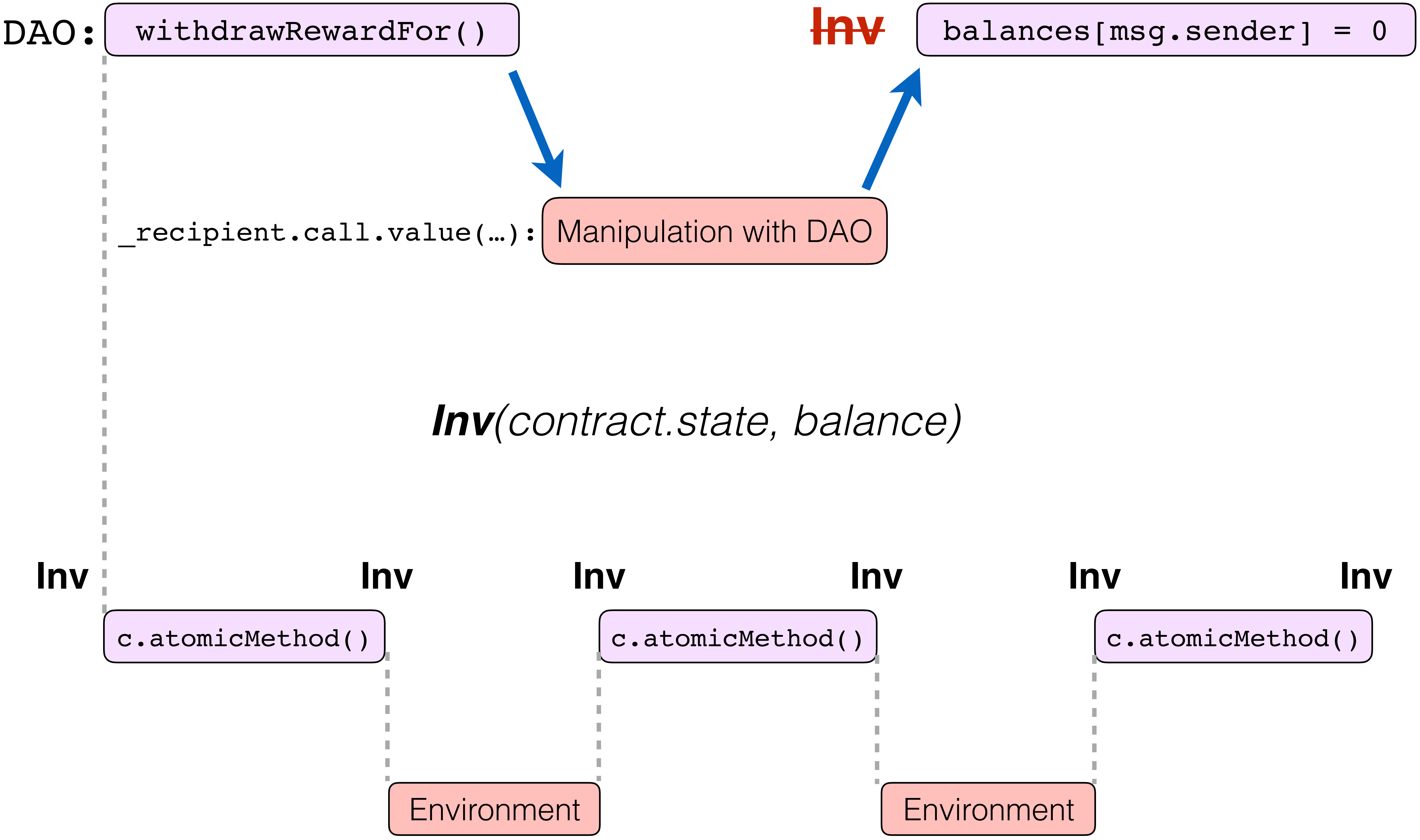
DAO:

withdrawRewardFor()

balances[msg.sender] = 0

_recipient.call.value(…):

Manipulation with DAO



DAO: `withdrawRewardFor()`

~~Inv~~ `balances[msg.sender] = 0`

`_recipient.call.value(…):` Manipulation with DAO

***Inv****(contract.state, balance)*

Inv `c.atomicMethod()` Inv

Environment

Inv `c.atomicMethod()` Inv

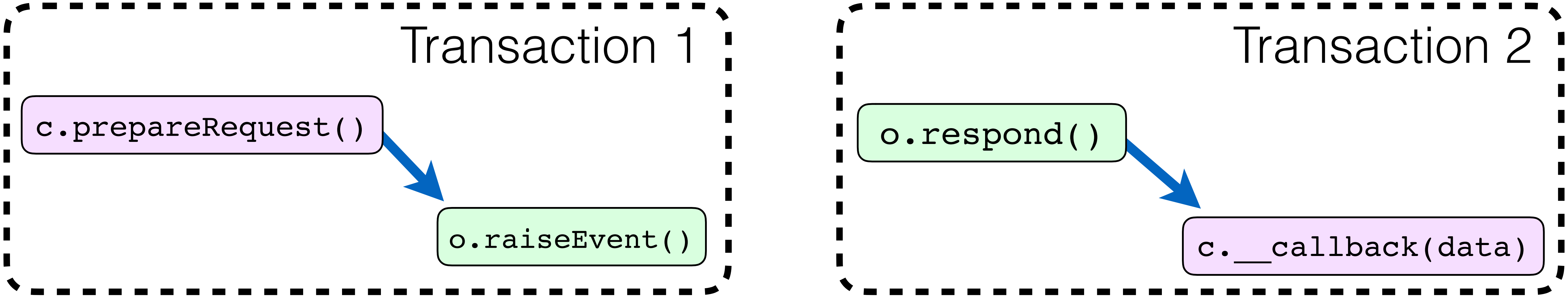
Environment

Inv `c.atomicMethod()` Inv

# *Accounts* using ***smart contracts*** in a blockchain
# are like
# *threads* using ***concurrent objects*** in shared memory.

| | | |
|---|---|---|
| contract state | — | object state |
| call/send | — | context switching |
| Reentrancy | — | (Un)cooperative multitasking |
| Invariants | — | Atomicity |

# Querying an Oracle



Transaction 1

c.prepareRequest()

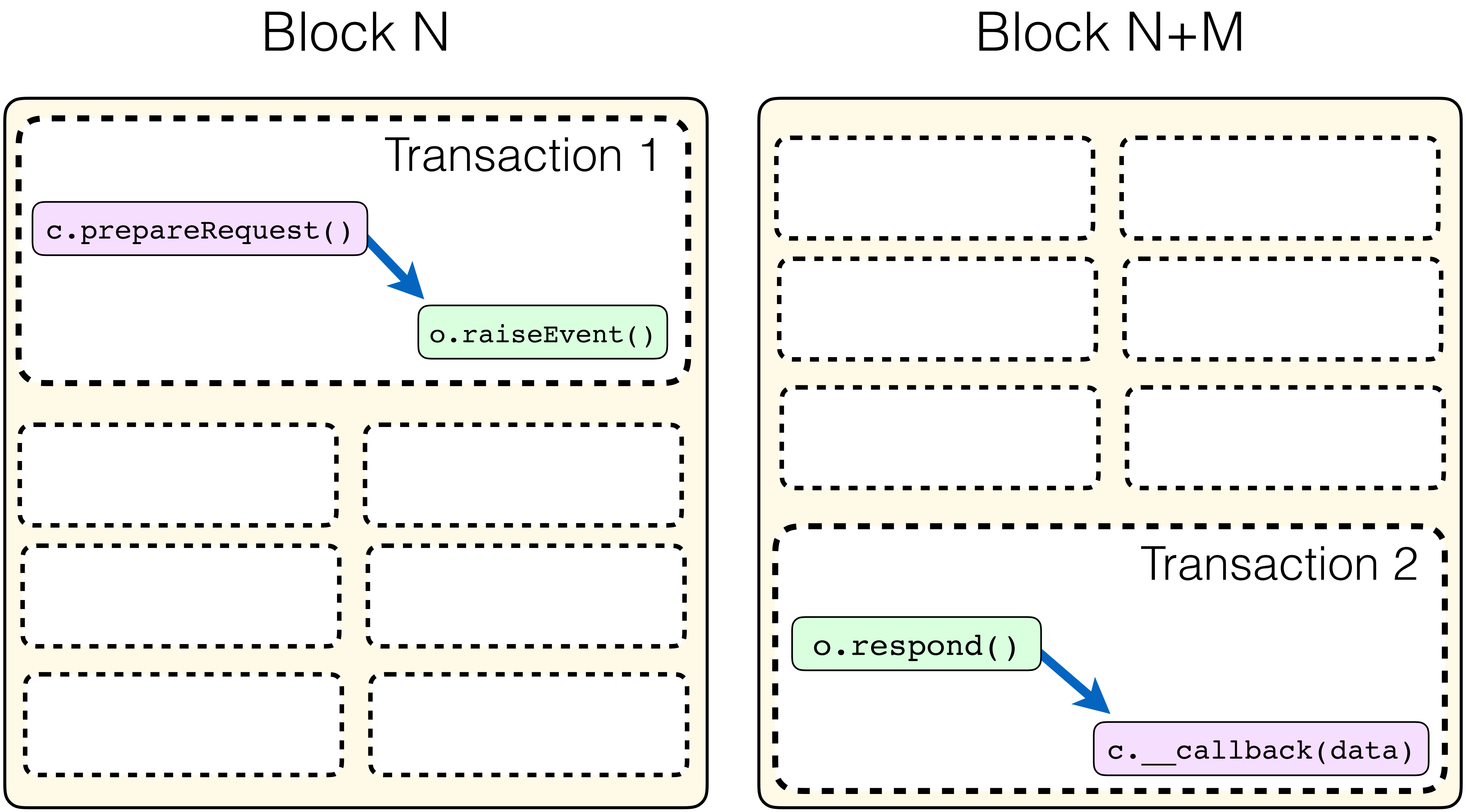
o.raiseEvent()

Transaction 2

o.respond()

c.__callback(data)

# Querying an Oracle



Block N

Transaction 1

c.prepareRequest()

o.raiseEvent()

Block N+M

Transaction 2

o.respond()

c.__callback(data)

# BlockKing via Oraclize

```
function enter() {
  if (msg.value < 50 finney) {
     msg.sender.send(msg.value);
     return;
  }
  warrior = msg.sender;
  warriorGold = msg.value;
  warriorBlock = block.number;
  bytes32 myid =
      oraclize_query(0,”WolframAlpha","random number between 1 and 9");
}
```

```
function __callback(bytes32 myid, string result) {
  if (msg.sender != oraclize_cbAddress()) throw;
  randomNumber = uint(bytes(result)[0]) - 48;
  process_payment();
}
```

# *Accounts* using ***smart contracts*** in a blockchain are like *threads* using ***concurrent objects*** in shared memory.

| | | |
|---|---|---|
| contract state | — | object state |
| call/send | — | context switching |
| Reentrancy | — | (Un)cooperative multitasking |
| Invariants | — | Atomicity |
| Non-determinism | — | data races |

# Reasoning about High-level Behavior of Contracts

(as of Concurrent Objects)

# Temporal Properties

$$Q\ \textit{since}\ P \overset{\text{def}}{=} \forall\, s\ s',\ \ s \rightarrow_{C}^{*} s',\ P(s) \Rightarrow Q(s, s')$$

- “Token price only goes up”;
- “No payments accepted after the quorum is reached”;
- “No changes can be made after locking”;
- “Consensus results are irrevocable”;
- *etc*.

# Work in Progress

- A Coq-based DSL for formally defining high-level contract behavior as of a “concurrent object”;
- Definitions of generic semantic contract properties;
- *Formal proofs* for several case studies (in Coq);
- Reasoning about contract/object composition;
- A verified compiler from the DSL to EVM;
- A compiler from Solidity to the DSL;

# To take away

*Accounts* using ***smart contracts*** in a blockchain
are like
*threads* using ***concurrent objects*** in shared memory.

- Understanding *intra*- and *inter*-transactional behavior;
- Detecting *atomicity violations* and *data races*;
- Repurposing *existing* verification ideas;

Thanks!