# Implementing and Mechanically Verifying Smart Contracts

Ilya Sergey

ilyasergey.net

# Smart Contracts

- *Stateful mutable* objects replicated via a (Byzantine) consensus protocol
- State typically involves a stored amount of *funds/currency*
- One or more entry points: invoked *reactively* by a client *transaction*
- Main usages:
    - crowdfunding and ICO
    - multi-party accounting
    - voting and arbitration
    - puzzle-solving games with distribution of rewards
- Supporting platforms: **Ethereum**, **Tezos, Zilliqa, EOS**, ...

```
contract Accounting {
  /* Define contract fields */
  address owner;
  mapping (address => uint) assets;

  /* This runs when the contract is executed */
  function Accounting(address _owner) {
    owner = _owner;
  }

  /* Sending funds to a contract */
  function invest() returns (string) {
    if (assets[msg.sender].initialized()) { throw; }
    assets[msg.sender] = msg.value;
    return "You have given us your money";
  }
}
```

Mutable fields

Constructor

Entry point

- `msg` argument is implicit
- funds accepted implicitly
- can be called as a function from another contract

```
contract Accounting {
  /* Define contract fields */
  address owner;
  mapping (address => uint) assets;

  /* This runs when the contract is executed */
  function Accounting(address _owner) {
    owner = _owner;
  }

  /* Sending funds to a contract */
  function invest() returns (string) {
    if (assets[msg.sender].initialized()) { throw; }
    assets[msg.sender] = msg.value;
    return "You have given us your money";
  }

  function stealMoney() {
    if (msg.sender == owner) { owner.send(this.balance) }
  }
}
```

# Misconceptions about Smart Contracts

| | |
|---|---|
| Deployed in a low-level language | Uniform compilation target |
| Must be *Turing-complete* | Run arbitrary computations |
| Code is law | What else if not the code? |

# Misconceptions about Smart Contracts

| | |
|---|---|
| Deployed in a low-level language | **Infeasible** audit and verification |
| Must be *Turing-complete* | **DoS** attacks, cost semantics, **exploits** |
| Code is law | **Cannot** be amended once deployed |

# What about High-Level Languages?

```
contract Accounting {
  /* Define contract fields */
  address owner;
  mapping (address => uint) assets;

  /* This runs when the contract is executed */
  function Accounting(address _owner) {
    owner = _owner;
  }

  /* Sending funds to a contract */
  function invest() returns (string) {
    if (assets[msg.sender].initialized()) { throw; }
    assets[msg.sender] = msg.value;
    return "You have given us your money";
  }
}
```

## Ethereum's **Solidity**

- JavaScript-like syntax
- *Calling* a function = *sending* funds
- *General* recursion and loops
- *Reflection*, *dynamic* contract creation
- Lots of *implicit* conventions
- No *formal* semantics

# el Languages?

Bernhard Mueller Follow
Security Engineer @ConsenSys
Nov 8, 2017 · 3 min read

## What caused the latest $100 million Ethereum smart contract bug

On November 6th, a user playing with the Pari
contract "accidentally" triggered its kill() funct
funds on all Parity multisig wallets linked to th
early estimates this might have made more tha
inaccessible (update: in the meantime, that nu
million).

```
/* Sending funds to a contract */
```

## Solidity optimizer bug

Posted by Martin Swende on May 3rd, 2017.

A bug in the Solidity optimizer was reported through the Ethereum Foundation Bounty program, by Christoph Jentzsch. This bug is patched as of 2017-05-03, with the release of Solidity 0.4.11.

r optimizes on constants in the byte code. By "byte
SH ed on the stack (not to be confused with Solidity

## List of Known Bugs

Below, you can find a JSON-formatted list of some of the known security-relevant bugs in the Solidity compiler. The file itself is hosted in the Github repository. The list stretches back as far as version 0.3.0, bugs known to be present only in versions preceding that are not listed.

# Sending a Message or Calling?

```
contract Accounting {
  /* Other functions */

  /* Sending funds to a contract */
  function invest() returns (string) {
    if (assets[msg.sender].initialized()) { throw; }
    assets[msg.sender] = msg.value;
    return "You have given us your money";
  }

  function withdrawBalance() {
    uint amount = assets[msg.sender];
    if (msg.sender.call.value(amount)() == false) {
      throw;
    }
    assets[msg.sender] = 0;
  }
}
```

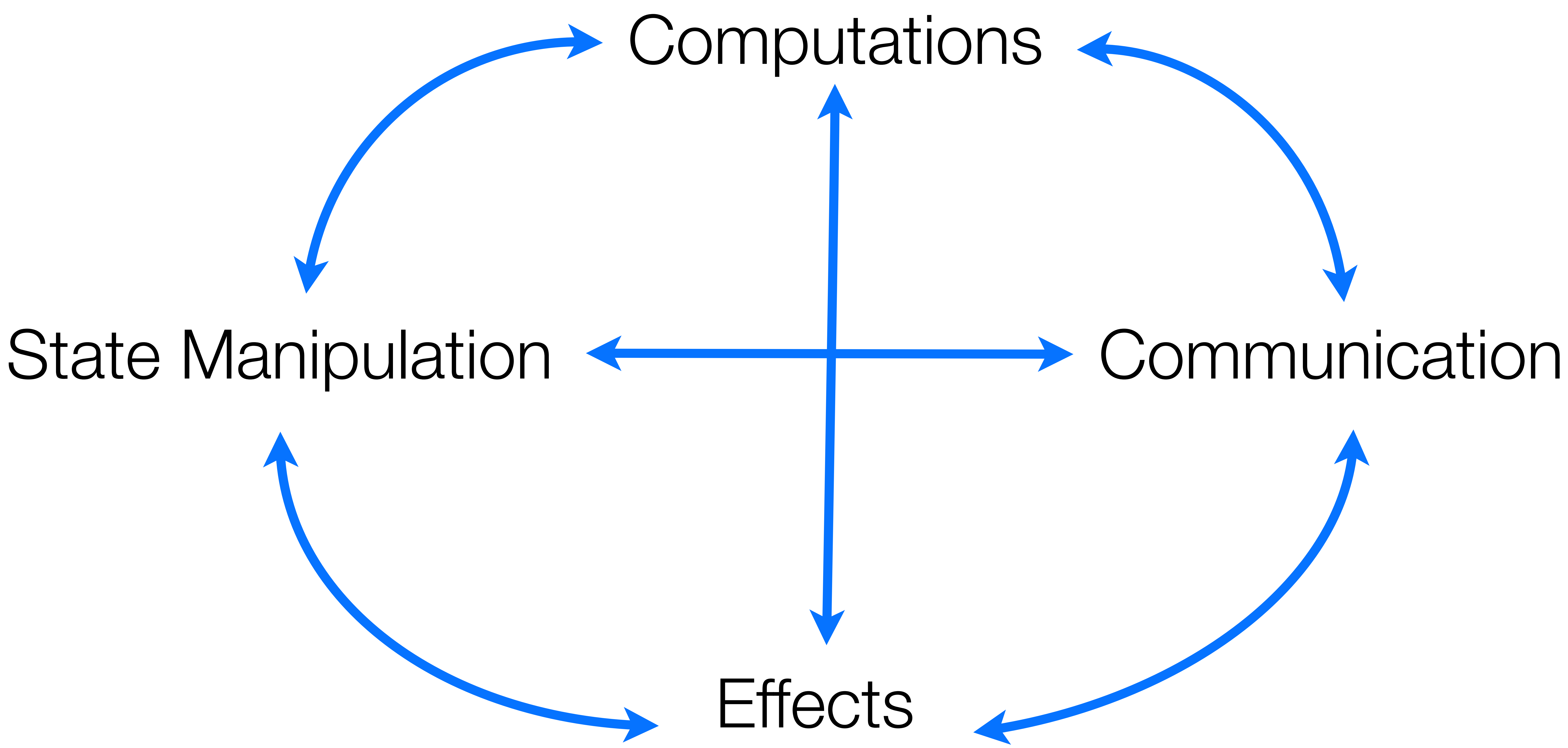
# Sending a Message or Calling?

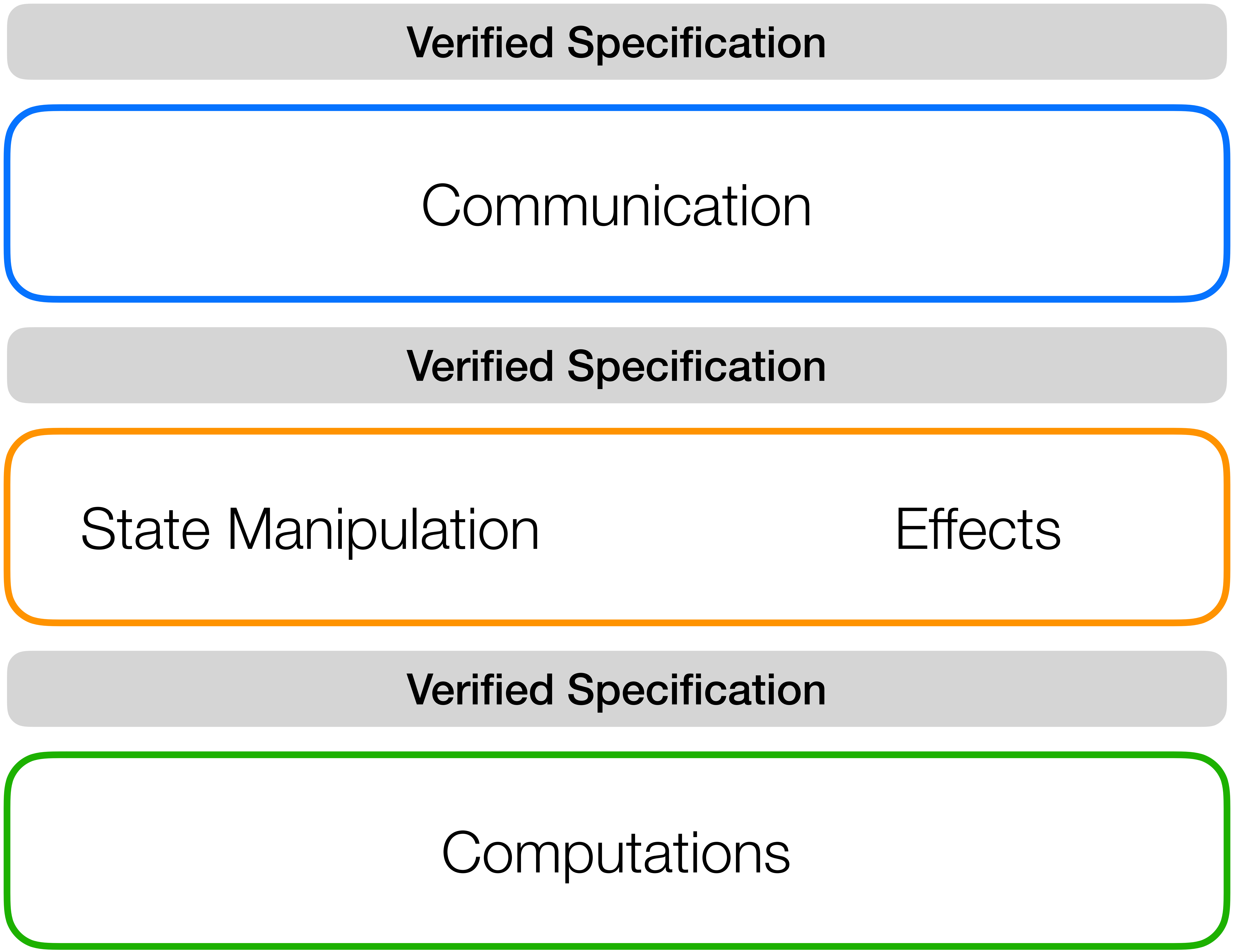
```
contract Accounting {
  /* Other functions */

  /* Sending funds to a contract */
  function invest() returns (string) {
    if (assets[msg.sender].initialized()) { throw; }
    assets[msg.sender] = msg.value;
    return "You have given us your money";
  }

  function withdrawBalance() {
    uint amount = assets[msg.sender];
    if (msg.sender.call.value(amount)() == false) {
      throw;
    }
    assets[msg.sender] = 0;
  }
}
```

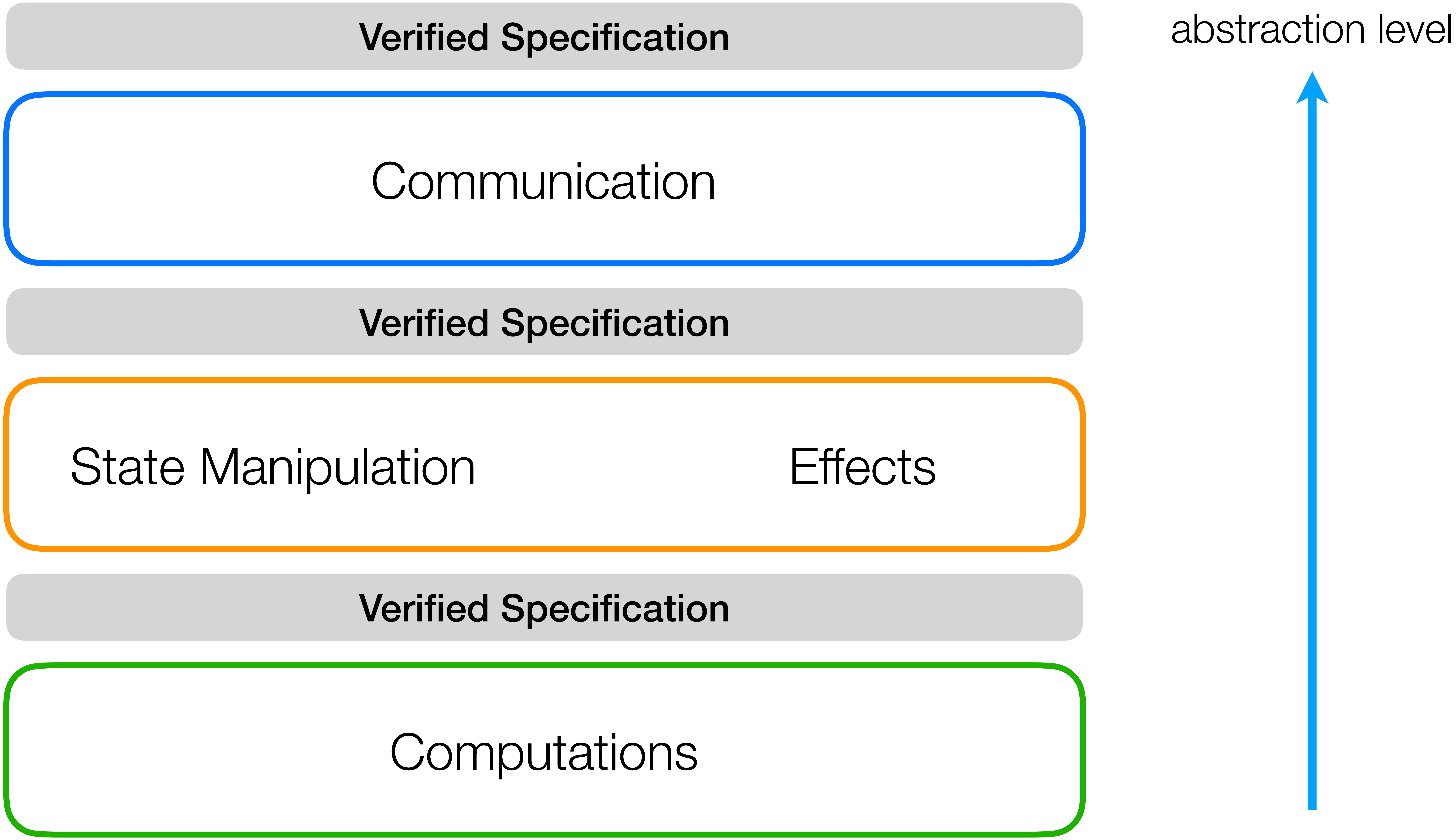
Can *reenter* and withdraw **again**

# Smart Contracts in a Nutshell

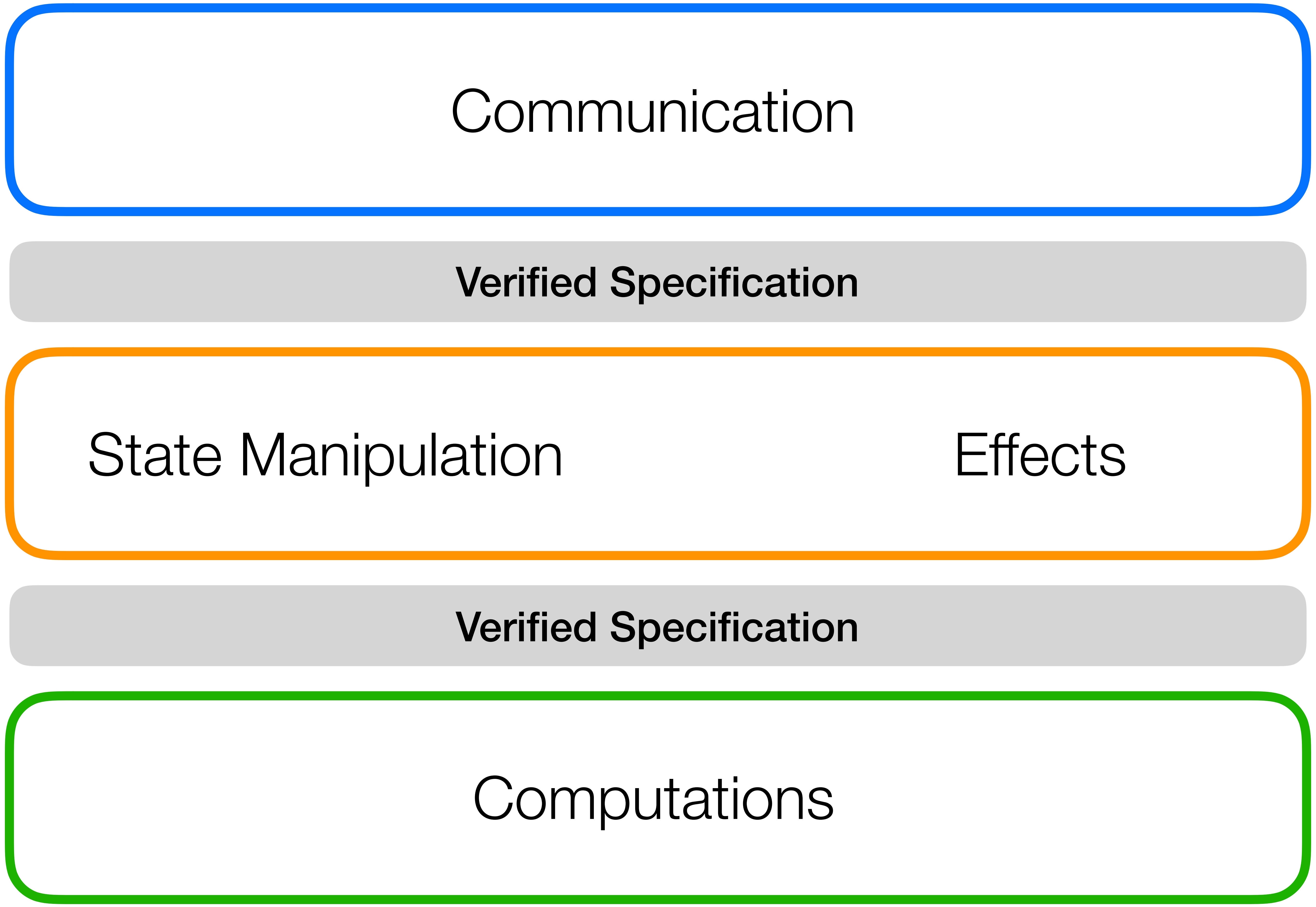
| | |
|---|---|
| Computations | self-explanatory |
| State Manipulation | changing contract's fields |
| Effects | accepting funds, logging events |
| Communication | sending funds, calling other contracts |


Computations
State Manipulation
Communication
Effects


Verified Specification
Communication
Verified Specification
State Manipulation
Effects
Verified Specification
Computations


Verified Specification
Communication
Verified Specification
State Manipulation
Effects
Verified Specification
Computations
abstraction level

# Scilla


Communication
Verified Specification
State Manipulation
Effects
Verified Specification
Computations

# Scilla

## Smart Contract Intermediate-Level Language

| | |
|---|---|
| Principled model for computations | **System F** with small extensions |
| *Not* Turing-complete | Only *primitive recursion*/iteration |
| Explicit Effects | *State-transformer* semantics |
| Communication | Contracts are *autonomous actors* |

# Types

| | | | | |
|---|---|---|---|---|
| Primitive type | $P$ | ::= | `Int` | Integer |
| | | | `String` | String |
| | | | `Hash` | Hash |
| | | | `BNum` | Block number |
| | | | `Address` | Account address |
| Type | $T, S$ | ::= | $P$ | primitive type |
| | | | `Map` $P$ $T$ | map |
| | | | `Message` | message |
| | | | $T$ `->` $S$ | value function |
| | | | $\mathcal{D}\ \langle T_k \rangle$ | instantiated data type |
| | | | $\alpha$ | type variable |
| | | | **`forall`** $\alpha\,.\,T$ | polymorphic function |

# Expressions (pure)

| | | | | |
|---|---|---|---|---|
| Expression | $e$ | ::= | $f$ | simple expression |
| | | | **let** $x$ ⟨: $T$⟩ = $f$ **in** $e$ | let-form |
| Simple expression | $f$ | ::= | $l$ | primitive literal |
| | | | $x$ | variable |
| | | | { ⟨$entry$⟩$_k$ } | Message |
| | | | **fun** ($x$ : $T$) => $e$ | function |
| | | | **builtin** $b$ ⟨$x_k$⟩ | built-in application |
| | | | $x$ ⟨$x_k$⟩ | application |
| | | | **tfun** $\alpha$ => $e$ | type function |
| | | | @$x$ $T$ | type instantiation |
| | | | C ⟨ {⟨$T_k$⟩} ⟩ ⟨$x_k$⟩ | constructor instantiation |
| | | | **match** $x$ **with** ⟨ \| $sel_k$ ⟩ **end** | pattern matching |
| Selector | $sel$ | ::= | $pat$ => $e$ | |
| Pattern | $pat$ | ::= | $x$ | variable binding |
| | | | C ⟨$pat_k$⟩ | constructor pattern |
| | | | ( $pat$ ) | paranthesized pattern |
| | | | _ | wildcard pattern |
| Message entrry | $entry$ | ::= | $b$ : $x$ | |
| Name | $b$ | | | identifier |

# Structural Recursion in Scilla

## Natural numbers (not Ints!)

```
nat_rec : forall α . α -> (nat -> α -> α) -> nat -> α
```

Result type

Value for 0

constructing the next value

number of iterations

final result

# Example: Fibonacci Numbers

```
let fib = fun (n : Nat) =>
  let iter_nat = @ nat_rec (Pair Int Int) in
  let iter_fun =
    fun (n: Nat) => fun (res : Pair Int Int) =>
      match res with
      | And x y => let z = builtin add x y in
                   And {Int Int} z x
      end
    in
  let zero = 0 in
  let one = 1 in
  let init_val = And {Int Int} one zero in
  let res = iter_nat init_val iter_fun n in
  fst res
```

# Example: Fibonacci Numbers

```
let fib = fun (n : Nat) =>
  let iter_nat = @ nat_rec (Pair Int Int) in
  let iter_fun =
    fun (n: Nat) => fun (res : Pair Int Int) =>
      match res with
      | And x y => let z = builtin add x y in
                   And {Int Int} z x
      end
    in
  let zero = 0 in
  let one = 1 in
  let init_val = And {Int Int} one zero in
  let res = iter_nat init_val iter_fun n in
  fst res
```

Value for 0: (1, 0)

# Example: Fibonacci Numbers

```
let fib = fun (n : Nat) =>
  let iter_nat = @ nat_rec (Pair Int Int) in
  let iter_fun =
    fun (n: Nat) => fun (res : Pair Int Int) =>
      match res with
      | And x y => let z = builtin add x y in
                   And {Int Int} z x
      end
    in
  let zero = 0 in
  let one = 1 in
  let init_val = And {Int Int} one zero in
  let res = iter_nat init_val iter_fun n in
  fst res
```

Iteration

# Example: Fibonacci Numbers

```
let fib = fun (n : Nat) =>
  let iter_nat = @ nat_rec (Pair Int Int) in
  let iter_fun =
    fun (n: Nat) => fun (res : Pair Int Int) =>
      match res with
      | And x y => let z = builtin add x y in
                   And {Int Int} z x
      end
    in
  let zero = 0 in
  let one = 1 in
  let init_val = And {Int Int} one zero in
  let res = iter_nat init_val iter_fun n in
  fst res
```

(x, y) → (x + y, x)

# Example: Fibonacci Numbers

```
let fib = fun (n : Nat) =>
  let iter_nat = @ nat_rec (Pair Int Int) in
  let iter_fun =
    fun (n: Nat) => fun (res : Pair Int Int) =>
      match res with
      | And x y => let z = builtin add x y in
                   And {Int Int} z x
      end
    in
  let zero = 0 in
  let one = 1 in
  let init_val = And {Int Int} one zero in
  let res = iter_nat init_val iter_fun n in
  fst res
```

The result of iteration is a *pair of integers*

# Example: Fibonacci Numbers

```
let fib = fun (n : Nat) =>
  let iter_nat = @ nat_rec (Pair Int Int) in
  let iter_fun =
    fun (n: Nat) => fun (res : Pair Int Int) =>
      match res with
      | And x y => let z = builtin add x y in
                   And {Int Int} z x
      end
    in
  let zero = 0 in
  let one = 1 in
  let init_val = And {Int Int} one zero in
  let res = iter_nat init_val iter_fun n in
  fst res
```

Iterate n times

# Example: Fibonacci Numbers

```
let fib = fun (n : Nat) =>
  let iter_nat = @ nat_rec (Pair Int Int) in
  let iter_fun =
    fun (n: Nat) => fun (res : Pair Int Int) =>
      match res with
      | And x y => let z = builtin add x y in
                   And {Int Int} z x
      end
    in
  let zero = 0 in
  let one = 1 in
  let init_val = And {Int Int} one zero in
  let res = iter_nat init_val iter_fun n in
  fst res
```

return *the first component* of the result pair

# Why Structural Recursion?

- Pros:
  - *All* programs *terminate*
  - Number of operations can be computed *statically* as a function of *input size*
- Cons:
  - Some functions cannot be implemented efficiently (e.g., QuickSort)
  - Cannot implement *Ackerman function* :(

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

# Statements (effectful)

| | | |
|---|---|---|
| s ::= | x <- f | read from mutable field |
| | f := x | store to a field |
| | x = e | assign a pure expression |
| | **match** x **with** ⟨pat => s⟩ **end** | pattern matching and branching |
| | x <- &B | read from blockchain state |
| | **accept** | accept incoming payment |
| | **send** ms | send list of messages |

# Statement Semantics

$$[\![s]\!] : BlockchainState \to Configuration \to Configuration$$

$BlockchainState$ — Immutable global data (block number *etc.*)

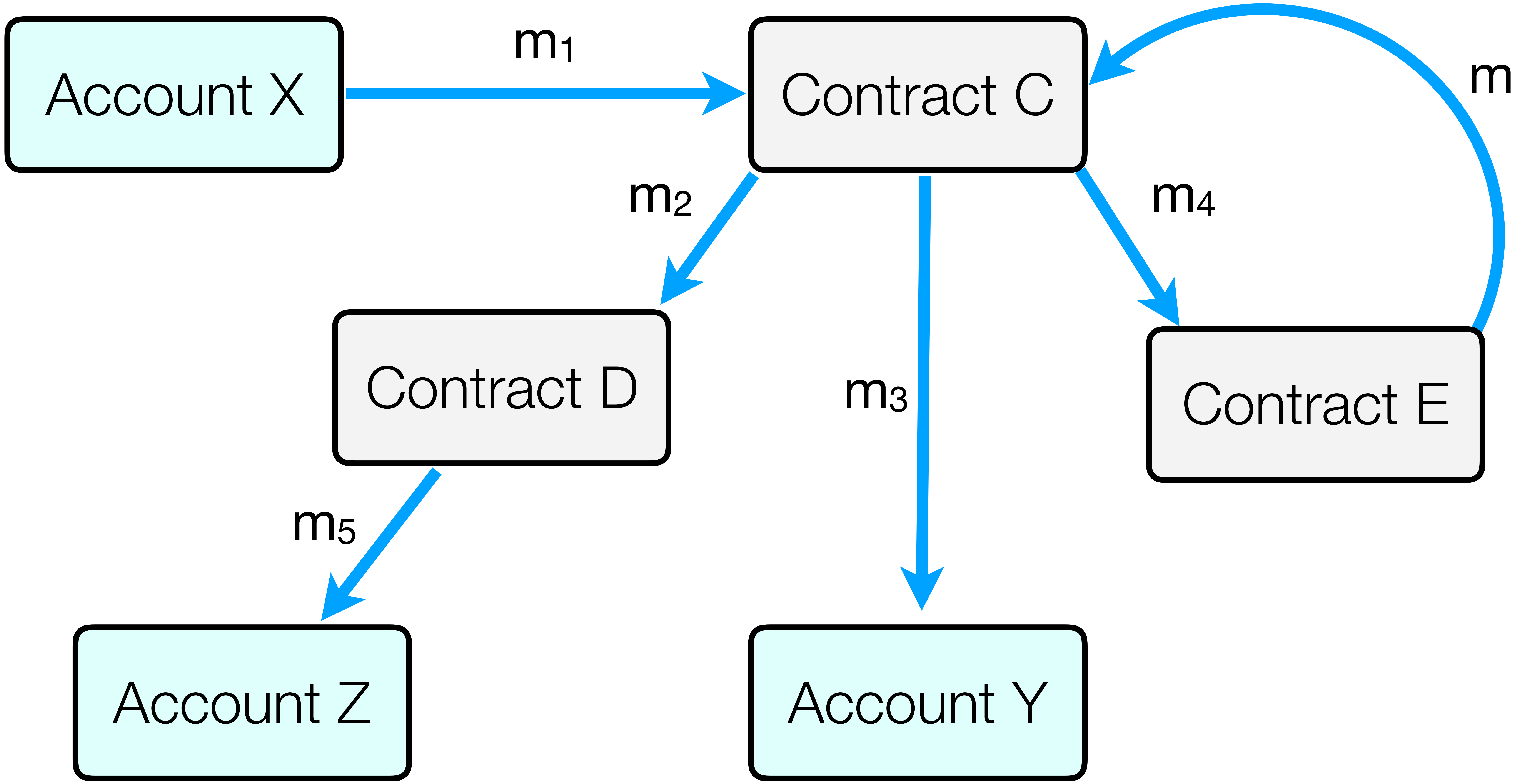
$$Configuration = Env \times Fields \times Balance \times Incoming \times Emitted$$

- $Env$: Immutable bindings
- $Fields$: Mutable fields
- $Balance$: Contract's own funds
- $Incoming$: Funds sent to contract
- $Emitted$: Messages to be sent

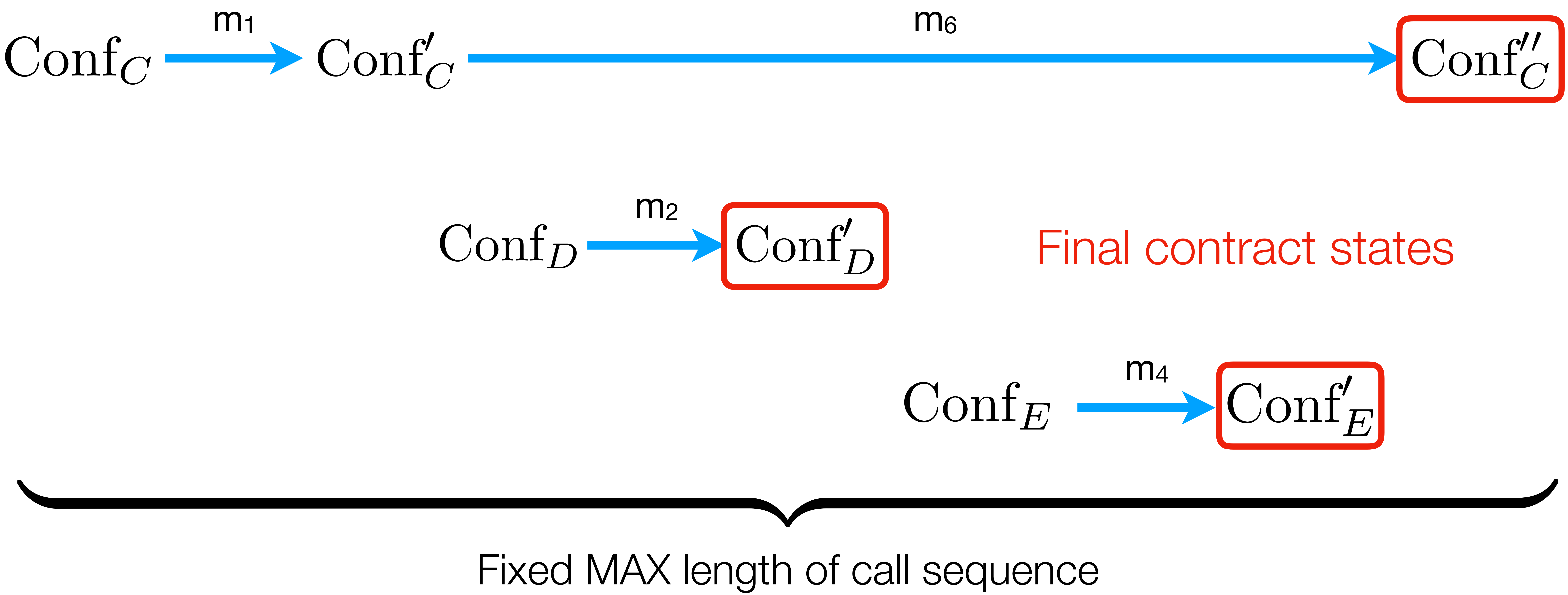
# Global Execution Model

Account X

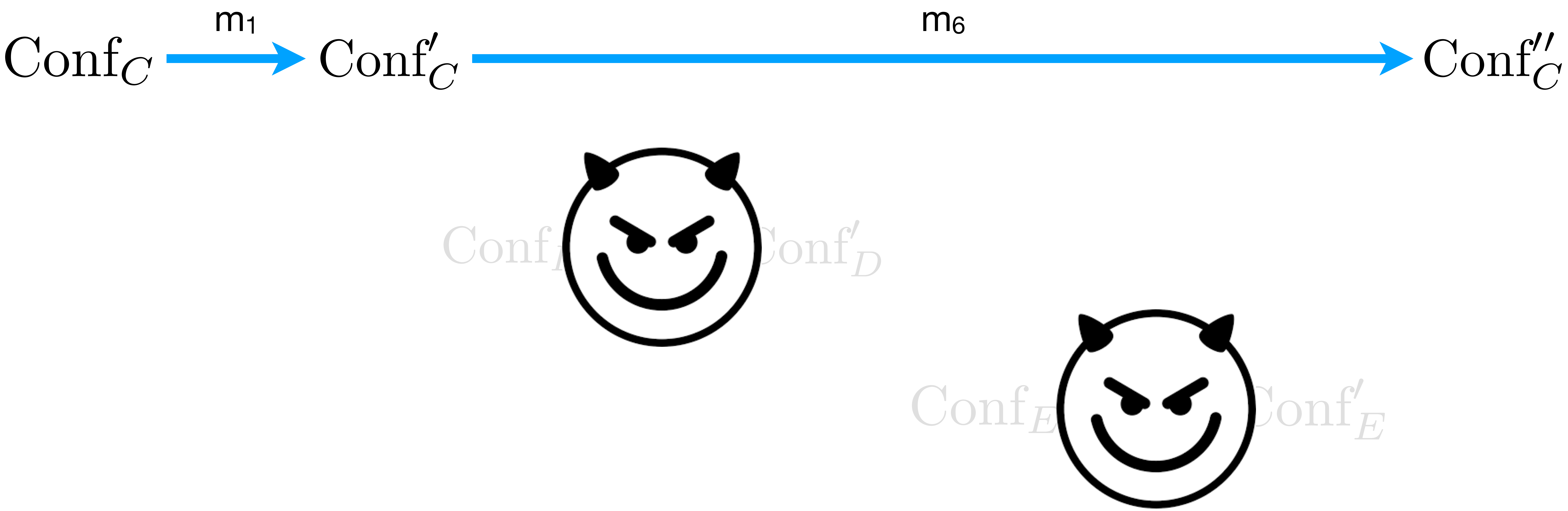
# Global Execution Model



# Global Execution Model



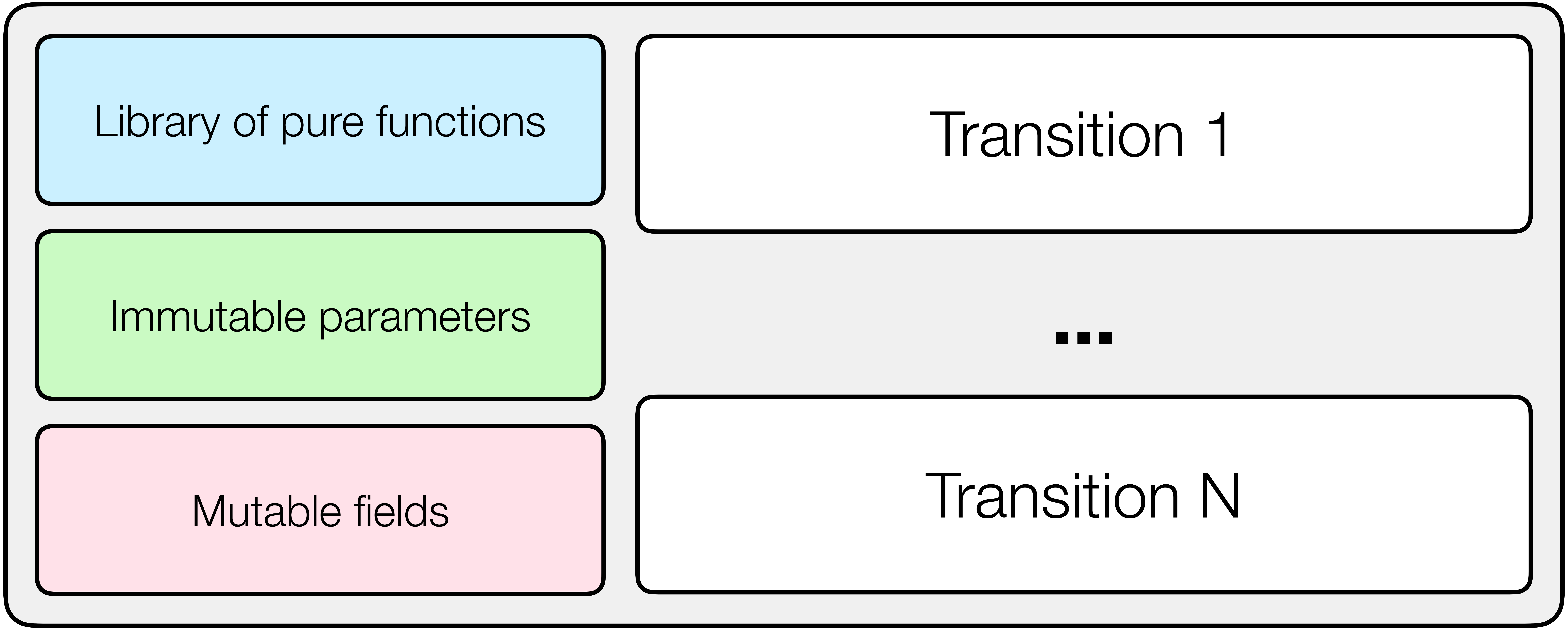
# Global Execution Model



# Putting it All Together

- Scilla contracts are (infinite) *State-Transition Systems*
- Interaction *between* contracts via sending/receiving *messages*
- Messages trigger (effectful) *transitions* (sequences of *statements*)
- A contract can *send messages* to other contracts via **`send`** statement
- Most computations are done via *pure expressions*, no storable closures
- Contract's state is immutable parameters, mutable fields, balance

# Contract Structure



# Working Example: *Crowdfunding* contract

- **Parameters**: campaign's *owner*, deadline (max block), funding *goal*
- **Fields**: *registry* of backers, *"campaign-complete"* boolean flag
- **Transitions**:
  - Donate money (when the campaign is active)
  - Get funds (as an owner, after the deadline, if the goal is met)
  - Reclaim donation (after the deadline, if the goal is not met)

```
transition Donate (sender: Address, amount: Int)
  blk <- & BLOCKNUMBER;
  in_time = blk_leq blk max_block;
  match in_time with
  | True  =>
    bs  <- backers;
    res = check_update bs sender amount;
    match res with
    | None =>
      msg  = {tag : Main; to : sender; amount : 0; code : already_backed};
      msgs = one_msg msg;
      send msgs
    | Some bs1 =>
      backers := bs1;
      accept;
      msg  = {tag : Main; to : sender; amount : 0; code : accepted_code};
      msgs = one_msg msg;
      send msgs
     end
  | False =>
    msg  = {tag : Main; to : sender; amount : 0; code : missed_dealine};
    msgs = one_msg msg;
    send msgs
  end
end
```

```
transition Donate (sender: Address, amount: Int)
  blk <- & BLOCKNUMBER;
  in_time = blk_leq blk max_block;
  match in_time with
  | True  =>
    bs  <- backers;
    res = check_update bs sender amount;
    match res with
    | None =>
      msg  = {tag : Main; to : sender; amount : 0; code : already_backed};
      msgs = one_msg msg;
      send msgs
    | Some bs1 =>
      backers := bs1;
      accept;
      msg  = {tag : Main; to : sender; amount : 0; code : accepted_code};
      msgs = one_msg msg;
      send msgs
     end
  | False =>
    msg  = {tag : Main; to : sender; amount : 0; code : missed_dealine};
    msgs = one_msg msg;
    send msgs
  end
end
```

Structure of the incoming message

```
transition Donate (sender: Address, amount: Int)
  blk <- & BLOCKNUMBER;
  in_time = blk_leq blk max_block;
  match in_time with
  | True  =>
    bs  <- backers;
    res = check_update bs sender amount;
    match res with
    | None =>
      msg  = {tag : Main; to : sender; amount : 0; code : already_backed};
      msgs = one_msg msg;
      send msgs
    | Some bs1 =>
      backers := bs1;
      accept;
      msg  = {tag : Main; to : sender; amount : 0; code : accepted_code};
      msgs = one_msg msg;
      send msgs
     end
  | False =>
    msg  = {tag : Main; to : sender; amount : 0; code : missed_dealine};
    msgs = one_msg msg;
    send msgs
  end
end
```

Reading from blockchain state

```
transition Donate (sender: Address, amount: Int)
  blk <- & BLOCKNUMBER;
  in_time = blk_leq blk max_block;
  match in_time with
  | True  =>
    bs  <- backers;
    res = check_update bs sender amount;
    match res with
    | None =>
      msg  = {tag : Main; to : sender; amount : 0; code : already_backed};
      msgs = one_msg msg;
      send msgs
    | Some bs1 =>
      backers := bs1;
      accept;
      msg  = {tag : Main; to : sender; amount : 0; code : accepted_code};
      msgs = one_msg msg;
      send msgs
     end
  | False =>
    msg  = {tag : Main; to : sender; amount : 0; code : missed_dealine};
    msgs = one_msg msg;
    send msgs
  end
end
```

Using pure library functions (defined above in the contract)

Manipulating with fields

```
transition Donate (sender: Address, amount: Int)
  blk <- & BLOCKNUMBER;
  in_time = blk_leq blk max_block;
  match in_time with
  | True  =>
    bs  <- backers;
    res = check_update bs sender amount;
    match res with
    | None =>
      msg  = {tag : Main; to : sender; amount : 0; code : already_backed};
      msgs = one_msg msg;
      send msgs
    | Some bs1 =>
      backers := bs1;
      accept;
      msg  = {tag : Main; to : sender; amount : 0; code : accepted_code};
      msgs = one_msg msg;
      send msgs
     end
  | False =>
    msg  = {tag : Main; to : sender; amount : 0; code : missed_dealine};
    msgs = one_msg msg;
    send msgs
  end
end
```

## Accepting incoming funds

```
transition Donate (sender: Address, amount: Int)
  blk <- & BLOCKNUMBER;
  in_time = blk_leq blk max_block;
  match in_time with
  | True  =>
    bs  <- backers;
    res = check_update bs sender amount;
    match res with
    | None =>
      msg  = {tag : Main; to : sender; amount : 0; code : already_backed};
      msgs = one_msg msg;
      send msgs
    | Some bs1 =>
      backers := bs1;
      accept;
      msg  = {tag : Main; to : sender; amount : 0; code : accepted_code};
      msgs = one_msg msg;
      send msgs
     end
  | False =>
    msg  = {tag : Main; to : sender; amount : 0; code : missed_dealine};
    msgs = one_msg msg;
    send msgs
  end
end
```

Creating and sending messages

```
transition Donate (sender: Address, amount: Int)
  blk <- & BLOCKNUMBER;
  in_time = blk_leq blk max_block;
  match in_time with
  | True  =>
    bs  <- backers;
    res = check_update bs sender amount;
    match res with
    | None =>
      msg  = {tag : Main; to : sender; amount : 0; code : already_backed};
      msgs = one_msg msg;
      send msgs
    | Some bs1 =>
      backers := bs1;
      accept;
      msg  = {tag : Main; to : sender; amount : 0; code : accepted_code};
      msgs = one_msg msg;
      send msgs
     end
  | False =>
    msg  = {tag : Main; to : sender; amount : 0; code : missed_dealine};
    msgs = one_msg msg;
    send msgs
  end
end
```

```
transition Donate (sender: Address, amount: Int)
  blk <- & BLOCKNUMBER;
  in_time = blk_leq blk max_block;
  match in_time with
  | True  =>
    bs  <- backers;
    res = check_update bs sender amount;
    match res with
    | None =>
      msg  = {tag : Main; to : sender; amount : 0; code : already_backed};
      msgs = one_msg msg;
      send msgs
    | Some bs1 =>
      backers := bs1;
      accept;
      msg  = {tag : Main; to : sender; amount : 0; code : accepted_code};
      msgs = one_msg msg;
      send msgs
     end
  | False =>
    msg  = {tag : Main; to : sender; amount : 0; code : missed_dealine};
    msgs = one_msg msg;
    send msgs
  end
end
```

Amount of own funds transferred in a message

Numeric code to inform the recipient

```
transition Donate (sender: Address, amount: Int)
  blk <- & BLOCKNUMBER;
  in_time = blk_leq blk max_block;
  match in_time with
  | True  =>
    bs  <- backers;
    res = check_update bs sender amount;
    match res with
    | None =>
      msg  = {tag : Main; to : sender; amount : 0; code : already_backed};
      msgs = one_msg msg;
      send msgs
    | Some bs1 =>
      backers := bs1;
      accept;
      msg  = {tag : Main; to : sender; amount : 0; code : accepted_code};
      msgs = one_msg msg;
      send msgs
     end
  | False =>
    msg  = {tag : Main; to : sender; amount : 0; code : missed_dealine};
    msgs = one_msg msg;
    send msgs
  end
end
```

# Demo

# Verifying Scilla Contracts

Scilla ⟷

Coq Proof Assistant

- Local properties (e.g., *"transition does not throw an exception"*)
- Invariants (e.g., *"balance is always strictly positive"*)
- Temporal properties (something good eventually happens)

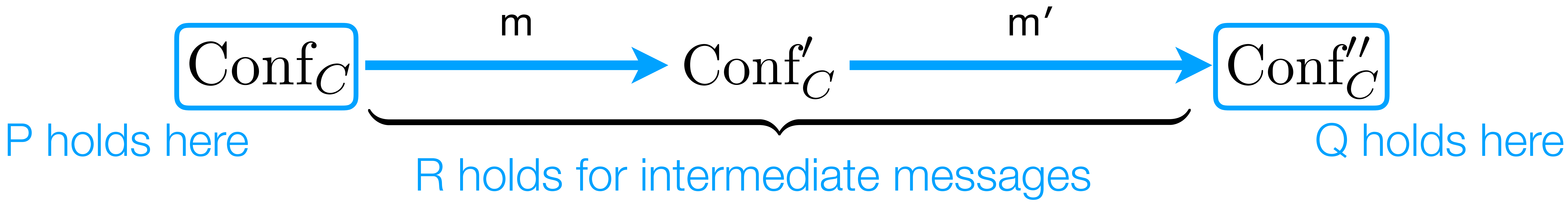
# Coq Proof Assistant

- *State-of-the* art verification framework
- Based on *dependently typed functional language*
- *Interactive* — requires a human in the loop
- Very small *trusted code base*
- Used to implement fully verified
  - *compilers*
  - *operating systems*
  - *distributed protocols (including blockchains)*

# Temporal Properties

Q *since* P as long R $\stackrel{\text{def}}{=}$

$\forall$ conf conf′, conf $\rightarrow_R^*$ conf′, P(conf) $\Rightarrow$ Q(conf, conf′)




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

# Temporal Properties

$$Q \textit{ since } P \text{ as long } R \stackrel{\text{def}}{=} \forall\, conf\ conf',\ \ conf \rightarrow_R^* conf',\ P(conf) \Rightarrow Q(conf, conf')$$

```
Definition since_as_long
           (P : conf → Prop)
           (Q : conf → conf → Prop)
           (R : bstate * message → Prop) :=
  ∀ sc conf conf',
    P st →
    (conf ⇝ conf' sc) ∧ (∀ b, b ∈ sc → R b) →
    Q conf conf'.
```

# Specifying properties of *Crowdfunding*

- **Lemma 1**: Contract *will always have enough balance* to refund everyone.
- **Lemma 2**: Contract will *not alter* its *contribution* records.
- **Lemma 3**: Each contributor will be refunded the right amount,
  *if the campaign fails.*

- **Lemma 2**: Contract will *not alter* its *contribution* records.

```
Definition donated (b : address) (d : amount) conf :=
  conf.backers(b) == d.
```

**b** donated amount **d**

```
Definition no_claims_from (b : address)
                          (q : bstate * message) :=
  q.message.sender != b.
```

**b** didn't try to claim

```
Lemma donation_preserved (b : address) (d : amount):
  since_as long (donated b d) (fun c c' => donated b d c')
                (no_claims_from b).
```

**b**'s records are preserved by the contract

# Demo

# Misconceptions, revisited

~~Need a low level language~~ — Need a language easy to reason about

~~Must be *Turing complete*~~ — Primitive recursion suffices in most cases

~~Code is law~~ — Code should abide by a specification

# What's next?

- Certified interpreter for Scilla contracts
- Compilation into an efficient back-end (LLVM, WASM)
- Certifications for *Proof-Carrying Code* (storable on a blockchain)
- *Automated Model Checking* smart contract properties
- PL support for *sharded contract executions*

# To Take Away

- Formal verification of *functional* and *temporal* properties of smart contracts requires a language with a clear separation of concerns
- Scilla: is a Smart Contract Intermediate-Level Language that provides it:
    - **Small**: builds on the polymorphic lambda-calculus with extensions.
    - **Principled**: separates computations, effects, and communication.
    - **Verifiable**: formal semantics and methodology for machine-assisted reasoning.

Thanks!

# Advertisement

- Do you want to work on *formal proofs* of correctness for *practicaldistributed systems* and *smart contracts* in **Coq**?


- Join the PhD program at National University of Singapore!
  - To start in **August 2019**, apply by **15 December 2018**
  - To start in **January 2020**, apply by **15 June 2019**

- Also, *postdoc positions* at Yale-NUS College are available starting **early 2019**.
- Get in touch with questions about topics and positions
  - Check **ilyaserey.net** for my contact details.