Scilla

Foundations for Verifiable Decentralised Computations on a Blockchain

Ilya Sergey

ilyasergey.net

Blockchain Consensus

$$\{tx_1, tx_3, tx_5, tx_4, tx_2\}$$

transactions
can be *anything*

- transforms a **set** of transactions into a *globally-agreed* **sequence**
- "distributed timestamp server" (Nakamoto 2008)

blockchain
consensus protocol

$$tx_5 \rightarrow tx_3 \rightarrow tx_4 \rightarrow tx_1 \rightarrow tx_2$$

Blockchain Consensus

$$\{tx_1, tx_3, tx_5, tx_4, tx_2\}$$

$$\downarrow$$

$$[tx_5, tx_3] \to [tx_4] \to [tx_1, tx_2]$$

$$\downarrow$$

$$tx_5 \to tx_3 \to tx_4 \to tx_1 \to tx_2$$

Blockchain Consensus

$$\{tx_1, tx_3, tx_5, tx_4, tx_2\}$$

$$[tx_5, tx_3] \leftarrow [tx_4] \leftarrow [tx_1, tx_2]$$

$$tx_5 \rightarrow tx_3 \rightarrow tx_4 \rightarrow tx_1 \rightarrow tx_2$$

Blockchain Consensus

$$\{tx_1, tx_3, tx_5, tx_4, tx_2\}$$

$$\downarrow$$

$$[] \leftarrow [tx_5, tx_3] \leftarrow [tx_4] \leftarrow [tx_1, tx_2]$$

GB = genesis block

$$\downarrow$$

$$tx_5 \rightarrow tx_3 \rightarrow tx_4 \rightarrow tx_1 \rightarrow tx_2$$

Transactions

- Executed *locally*, alter the *replicated* state.
- Simplest variant: *transferring funds* from *A* to *B,*
  consensus: *no* double spending.
- More interesting: deploying and executing $\underbrace{\textit{replicated computations}}_{\text{Smart Contracts}}$

Smart Contracts

- *Stateful mutable* objects replicated via a 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** (?), ...

```
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 Parit
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;
  }
}
```

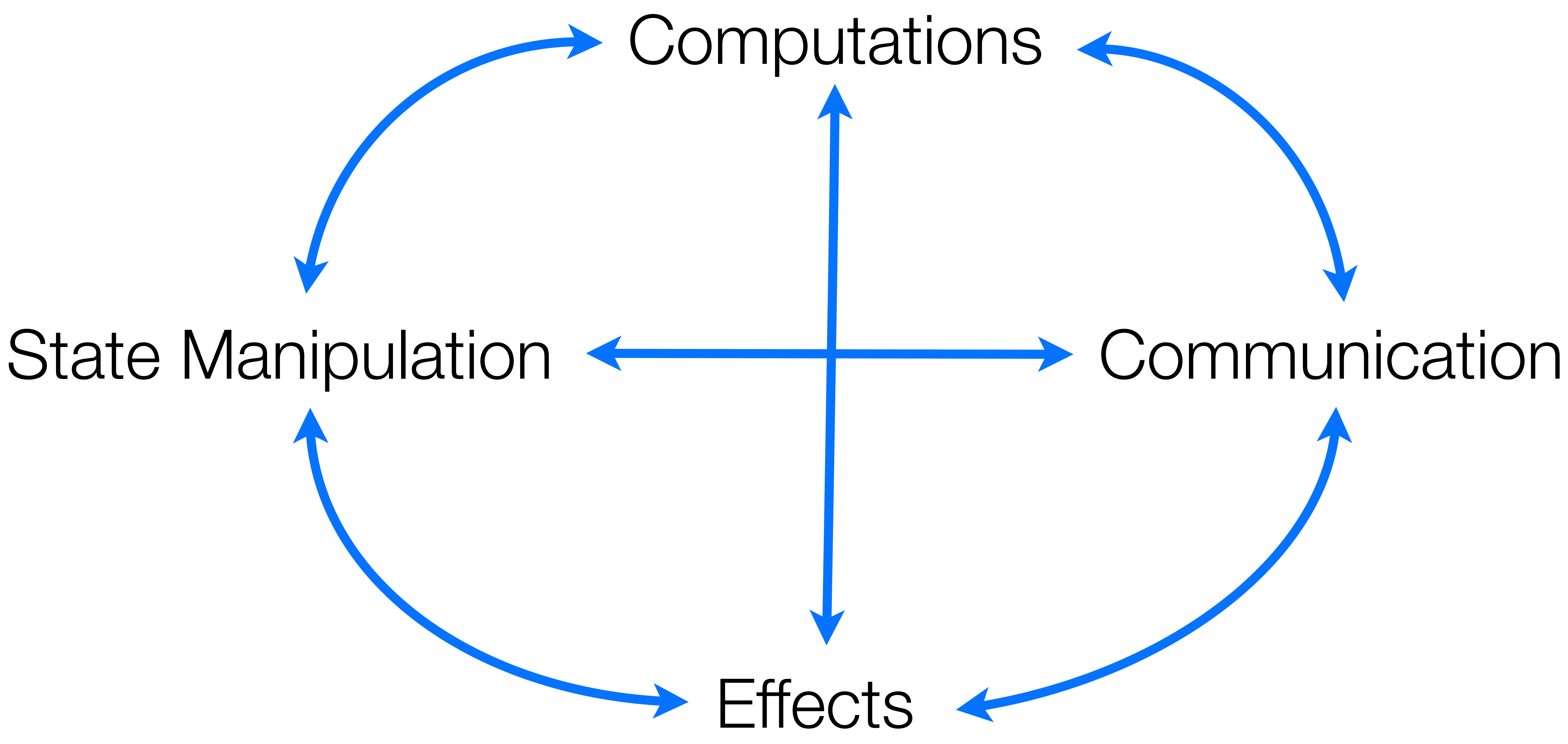
Sending a Message or Calling?

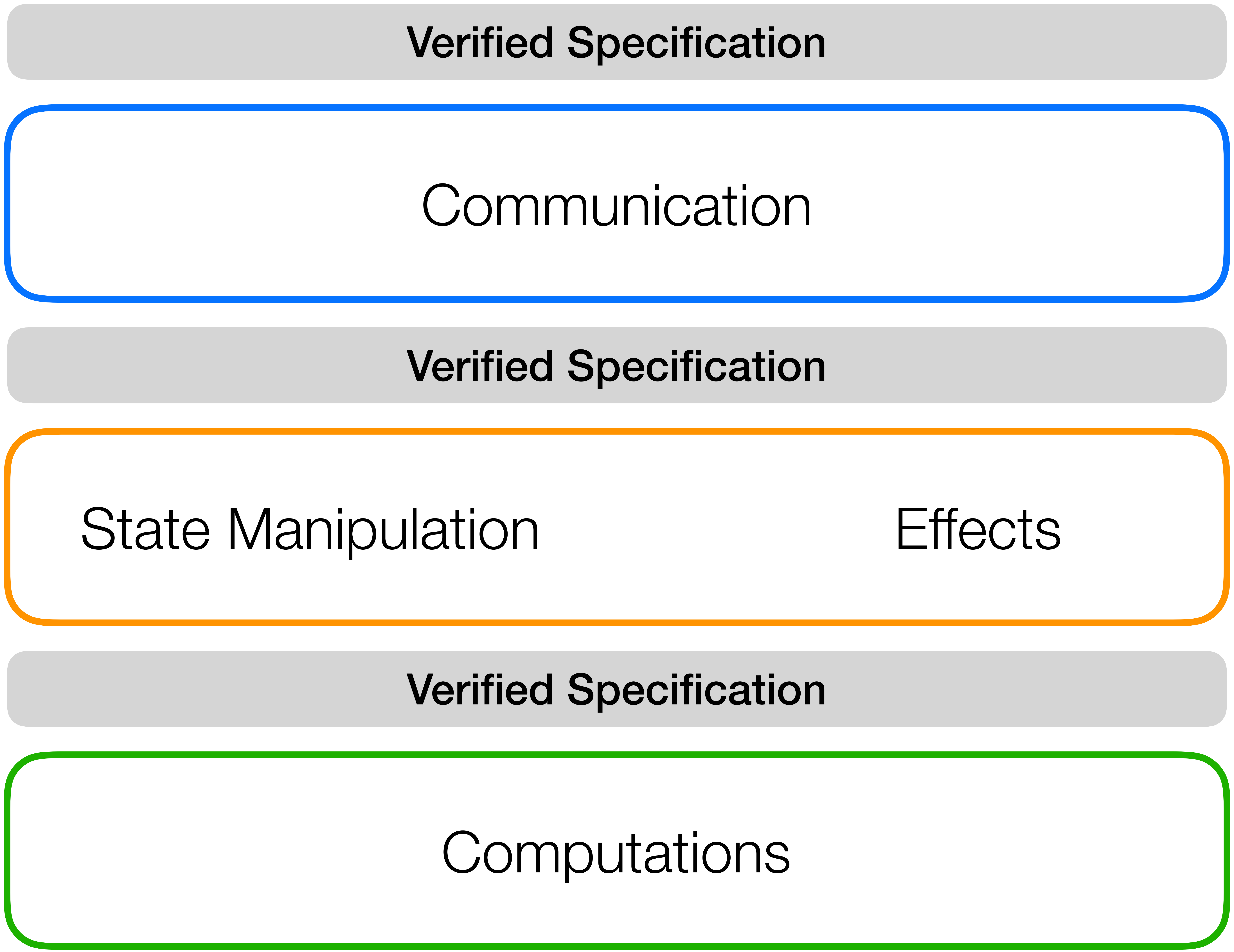
```
contract Accounting {
  /* Other functions */

  /* 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;
  }
}
```

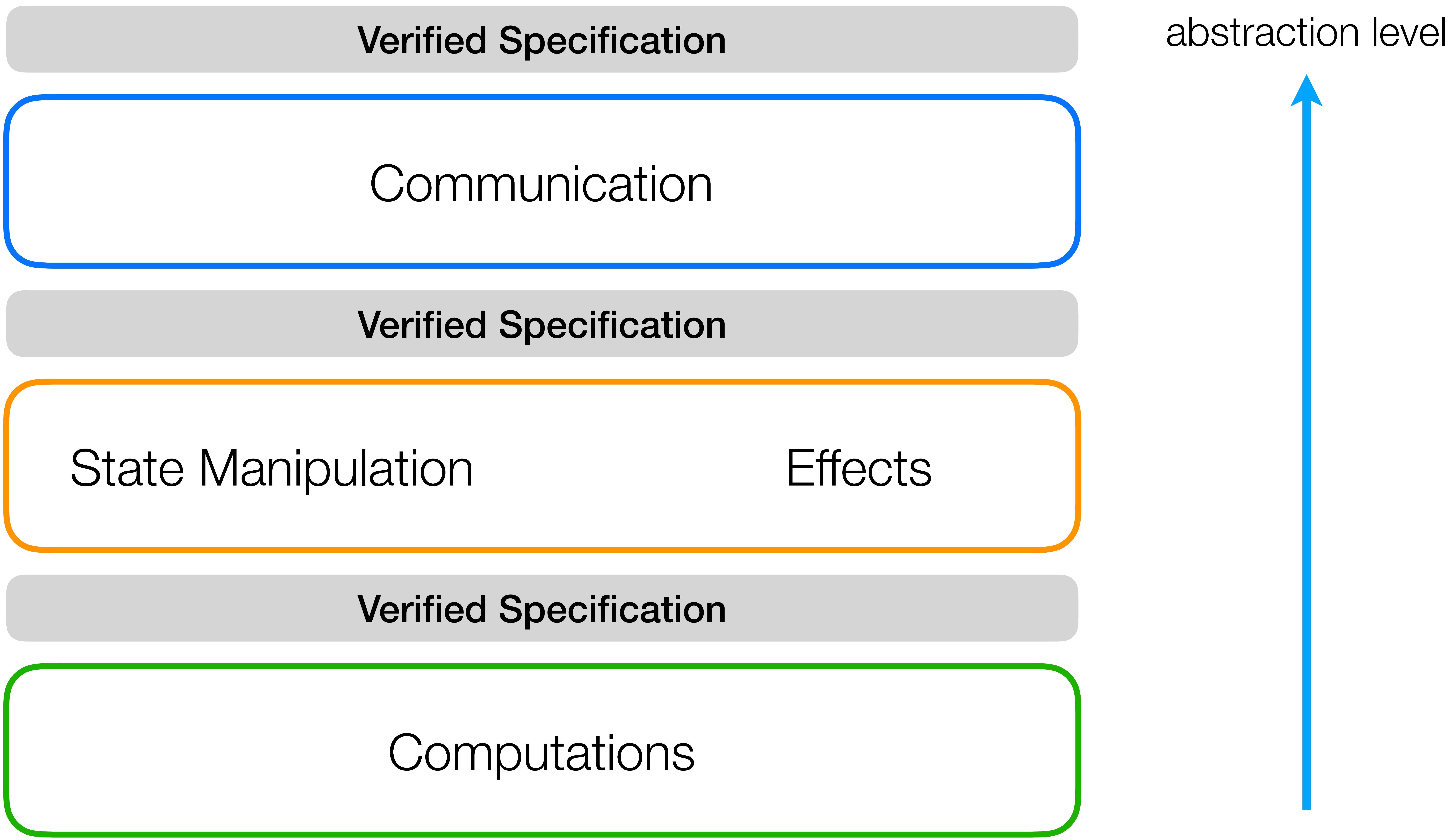
Can *reenter* and withdraw **again**

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
			α	type variable
			`forall` $\alpha\,.\,T$	polymorphic function

Expressions (pure)

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

Structural Recursion in Scilla

Natural numbers (not Ints!)

$$\texttt{nat\_rec} : \textbf{forall}\ \alpha . \alpha \to \underbrace{(\texttt{nat} \to \alpha \to \alpha)}_{\text{constructing the next value}} \to \texttt{nat} \to \alpha$$

Result type

Value for 0

number of iterations

constructing the next value

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

Structural Recursion with Lists

list_rec : **forall** α β. β -> $\underbrace{(\alpha \text{ -> list } \alpha \text{ -> } \beta \text{ -> } \beta)}_{\text{Iterator for non-empty list}}$ -> list α -> β

Element type

Result type

Value for Nil

argument list

argument list

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

$$[\![s]\!] : \mathit{BlockchainState} \rightarrow \mathit{Configuration} \rightarrow \mathit{Configuration}$$

$\mathit{BlockchainState}$ Immutable global data (block number *etc.*)

$$\mathit{Configuration} = \mathit{Env} \times \mathit{Fields} \times \mathit{Balance} \times \mathit{Incoming} \times \mathit{Emitted}$$

Immutable bindings

Mutable fields

Contract's own funds

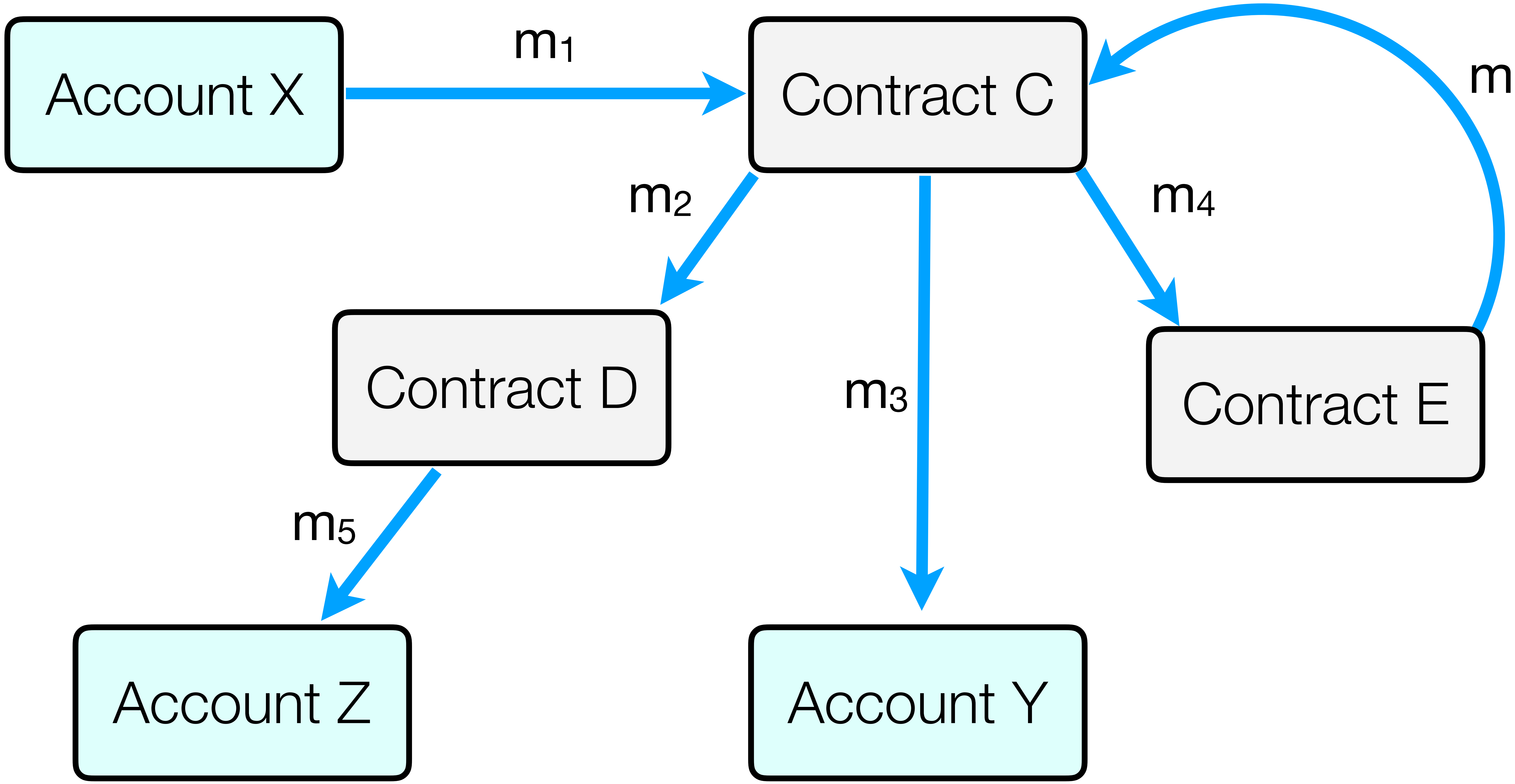
Funds sent to contract

Messages to be sent

Global Execution Model

Account X

Global Execution Model



Global Execution Model

$$\mathrm{Conf}_C \xrightarrow{m_1} \mathrm{Conf}'_C \xrightarrow{m_6} \boxed{\mathrm{Conf}''_C}$$

$$\mathrm{Conf}_D \xrightarrow{m_2} \boxed{\mathrm{Conf}'_D}$$

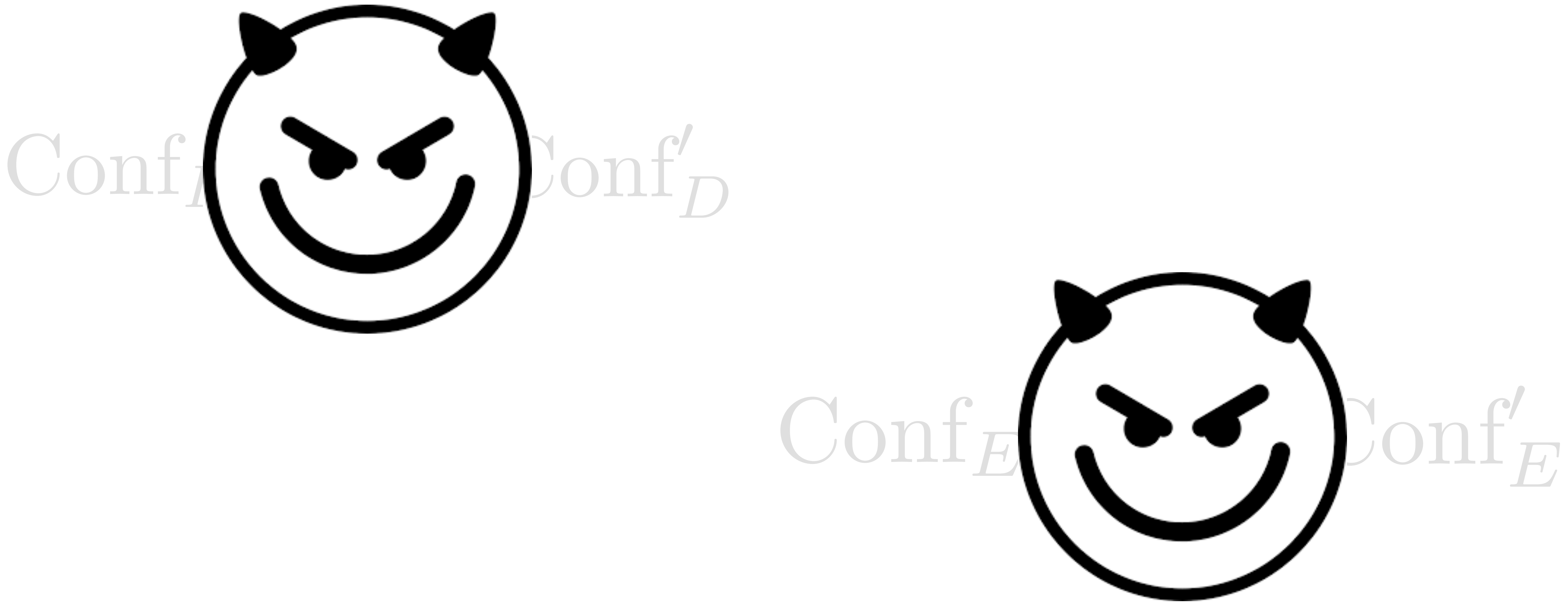
Final contract states

$$\mathrm{Conf}_E \xrightarrow{m_4} \boxed{\mathrm{Conf}'_E}$$

Fixed MAX length of call sequence

Global Execution Model

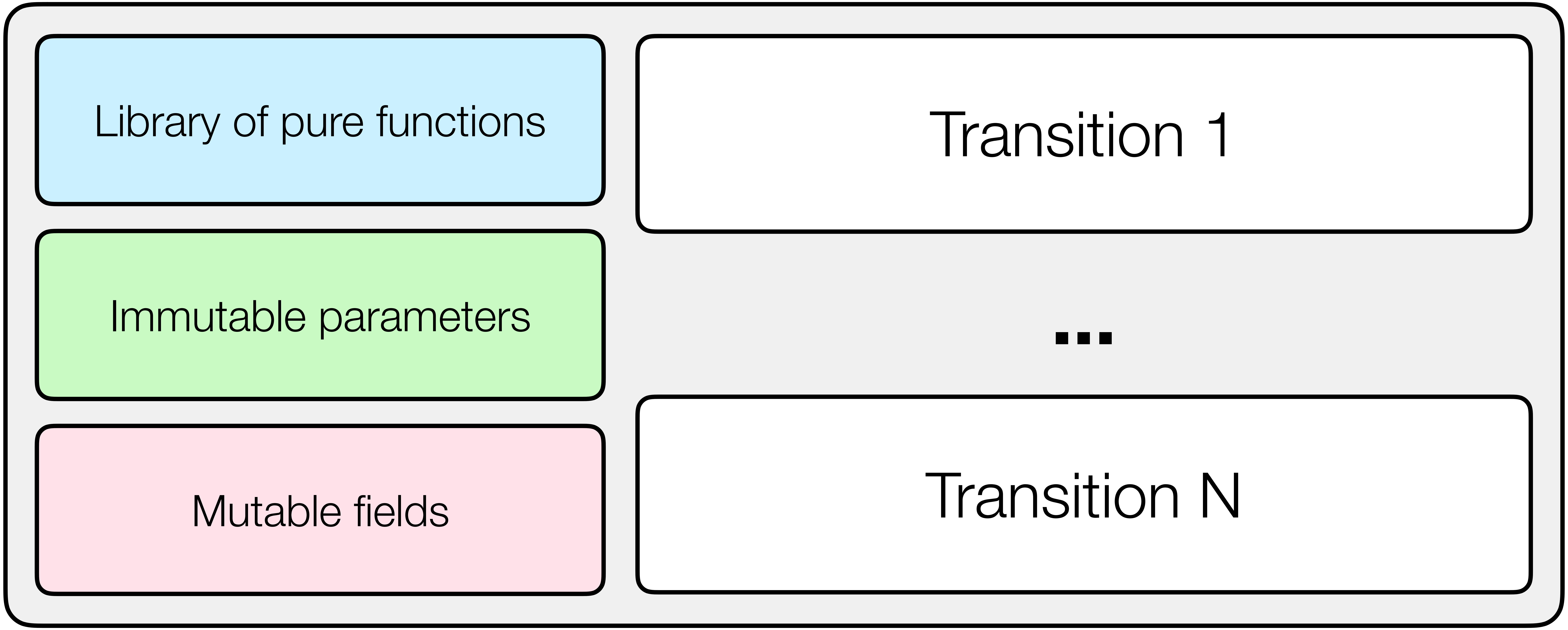
$$\mathrm{Conf}_C \xrightarrow{\text{m}_1} \mathrm{Conf}'_C \xrightarrow{\text{m}_6} \mathrm{Conf}''_C$$



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

```
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
```

Numeric code to inform the recipient

Demo

Scilla IDE
beta
Add Node
http://localhost:4201
(**)
(*   Associated library   *)
(**)
library Crowdfunding
let andb =
  fun (b : Bool) => fun (c : Bool) =>
    match b with
    | False => False
    | True => match c with
               | False => False
               | True => True
               end
    end

let orb =
  fun (b : Bool) => fun (c : Bool) =>
    match b with
    | True => True
    | False => match c with
               | False => False
               | True => True
               end
    end

let negb = fun (b : Bool) =>
  match b with
  | True => False
  | False => True
  end

let one_msg =
  fun (msg : Message) =>
   let nil_msg = Nil {Message} in
   Cons {Message} msg nil_msg

let check_update =
  fun (bs : Map Address Int) =>
    fun (sender : Address) =>
      fun (_amount : Int) =>
  let c = builtin contains bs sender in
  match c with
  | False =>
    let bs1 = builtin put bs sender _amount in
    Some {Map Address Int} bs1
  | True => None {Map Address Int}
  end

let blk_leq =
Initialization Parameters
owner
Address
490af4a007ce3d53d56
max_block
BNum
800
goal
Int
500
Add Param
Remove Param
Deploy Contract
0 pending transactions
0 confirmed transactions
Zilliqa Explorer
beta
Home
DS Blocks
Tx Blocks
Search
Address
Add Node
http://localhost:4201
# of Peers
20
# of DS Blocks
2
# of Tx Blocks
2
# of Transactions
0
Page Latency
5s
# of Txns in DS Epoch
0
# of Txns in Tx Epoch
0
Transaction Rate (tps)
0.00
Latest DS Blocks
BlockNum
Hash
1
F32F5B999642AB767453F131EC682885E24DAFAC590D4A9927FEBC4EEF185908
0
D47631EF571B848A8F17C705E51F3C1575E22306CFB6D45779CEE0687BDB4F98
See All
Latest Tx Blocks
BlockNum
Hash
1
87EC66E2AD18709BF6657629A0C295603A2BE8F3DF335AE7235DC31E7C5BDBC2
0
39A2343CFF2FA9E9612405150968E026AD3310B773D5424B4BADC144C5111E0C
See All
Latest Transactions
Transaction Hash
Newer
Older
About
Zilliqa © 2018
About
Zilliqa © 2018

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 $\stackrel{\text{def}}{=}$

$$\forall\, conf\ conf',\ \ conf \rightarrow_R^{*} conf',\ P(conf) \Rightarrow Q(conf, conf')$$

$$\boxed{\mathrm{Conf}_C} \underbrace{\xrightarrow{\ m\ } \mathrm{Conf}'_C \xrightarrow{\ m'\ }}_{\text{R holds for intermediate messages}} \boxed{\mathrm{Conf}''_C}$$

P holds here

Q holds here

- "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

Modeling Crowdfunding in COQ

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

To Take Away

Scilla: Smart Contract Intermediate-Level Language

- Small: builds on the *polymorphic lambda-calculus* with extensions.
- Principled: separates *computations*, *effects*, and *communication*.
- Verifiable: *formal semantics* and methodology for *machine-assisted reasoning*.

Work in Progress

- Integrating with an existing blockchain solution
- Compilation into an efficient back-end (LLVM)
- Certifications for *Proof-Carrying Code* (storable on a blockchain)
- *Automated Model Checking* smart contract properties
- PL support for *sharded contract executions*

Thanks!