The **Scilla** Journey: From Proof General to Thousands of Nodes

Ilya Sergey

ilyasergey.net

Yale NUS College

NUS National University of Singapore

zilliqa

Prologue

The Technology

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

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

$$\downarrow$$

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

$$\downarrow$$

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

Blockchain Consensus

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

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

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

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 *replicated computations*

Smart Contracts

Smart Contracts (Account Model)

- *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**, **Concordium**, **Libra**, **Cardano***,…

```
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) }
  }
}
```

The Givens of 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?

The Givens of Smart Contracts

Deployed in a low-level language	**Difficult** for audit and verification
Must be *Turing-complete*	Complex semantics, **exploits**
Code is law	One should understand the **code** to understand the **contract**

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**

What’s the Right Model
of thinking
about Smart Contracts?

Chapter I

The Analogy

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

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.

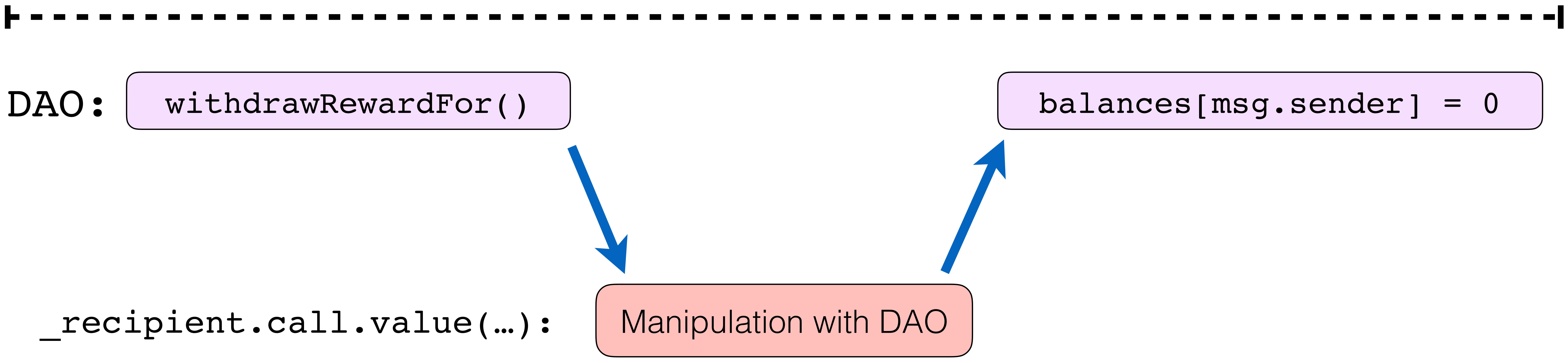
contract state	—	object state
call/send	—	context switching
Reentrancy	—	(Un)cooperative multitasking

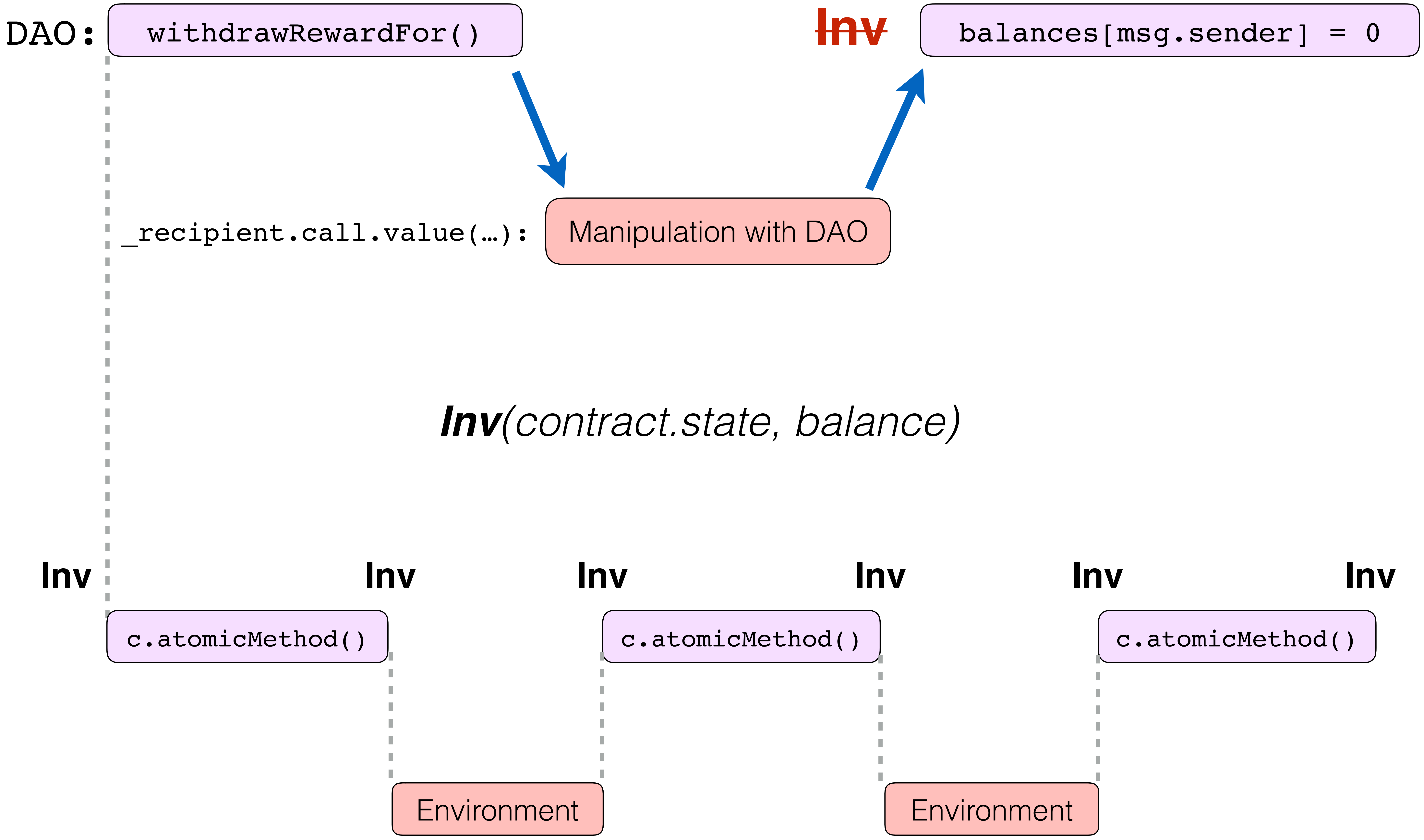
Reentrancy and multitasking

```
  // 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;
}
```



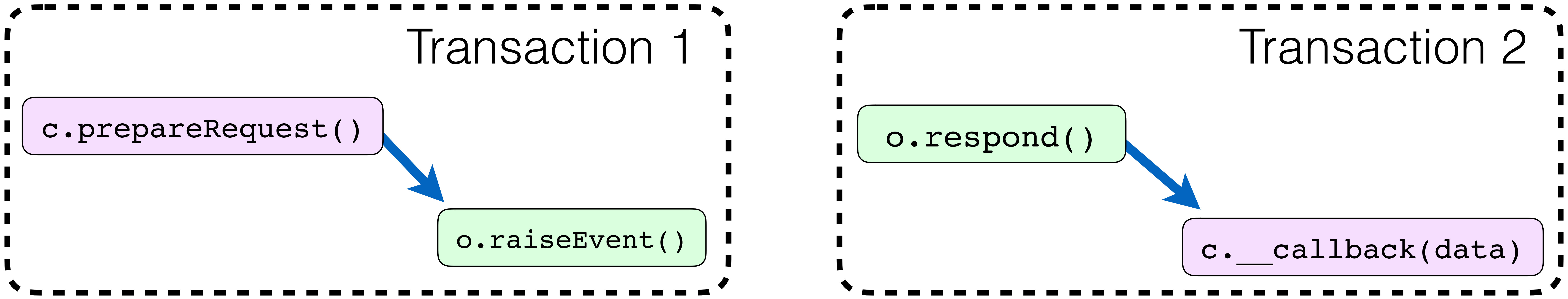


DAO:
withdrawRewardFor()
Inv
balances[msg.sender] = 0
_recipient.call.value(…):
Manipulation with DAO
Inv(contract.state, balance)
Inv
Inv
Inv
Inv
Inv
Inv
c.atomicMethod()
c.atomicMethod()
c.atomicMethod()
Environment
Environment

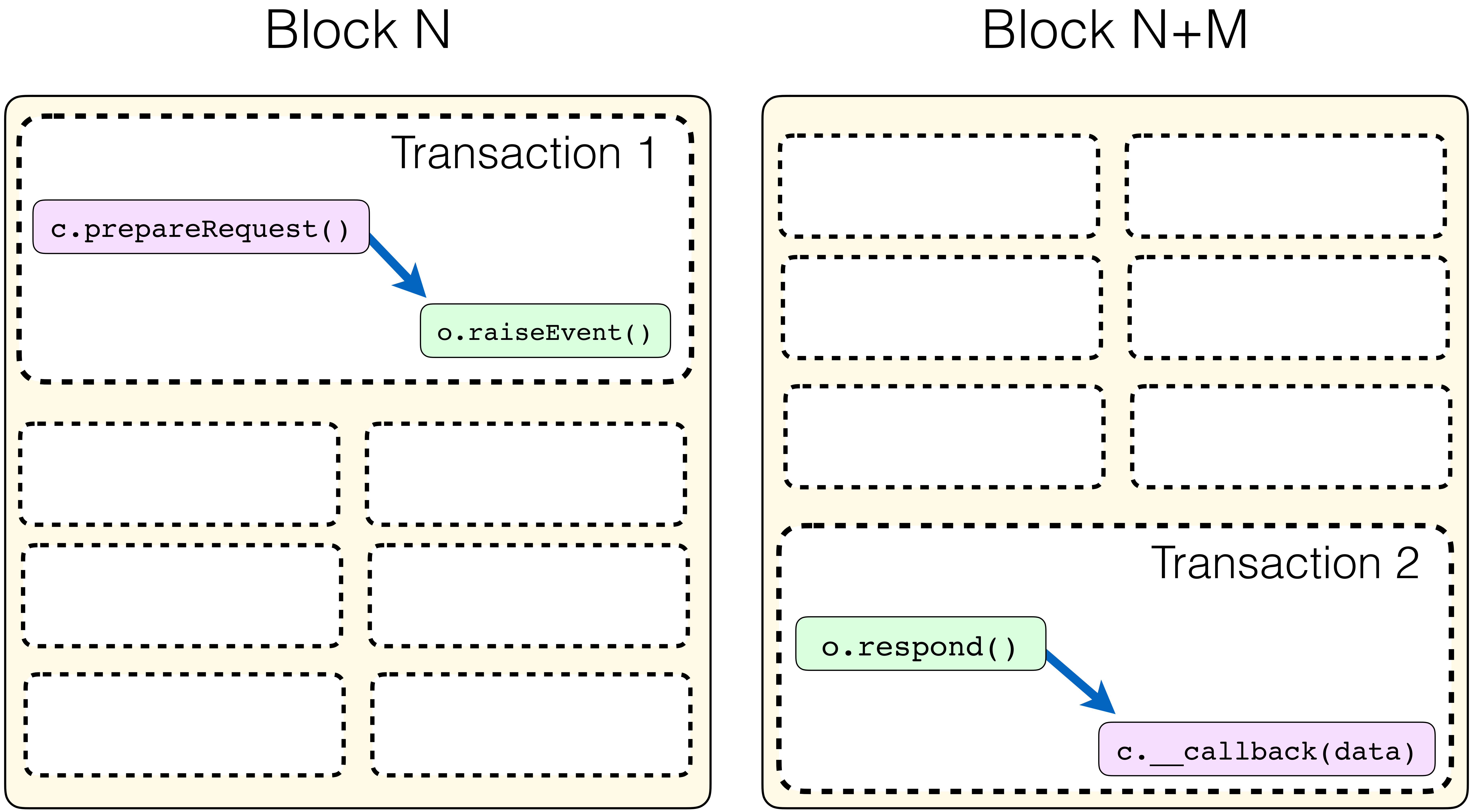
Accounts using ***smart contracts*** in a blockchain
are like
threads using ***concurrent objects*** in shared memory.

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

Querying an Oracle



Querying an Oracle



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

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

Exploiting the Laws of Order i

Aashish Kolluri
School of Computing, NUS
Singapore

Ivica Nikolic
School of Computing, N
Singapore

Finding The Greedy, Prodigal, and Suicidal Contracts at Scale

Aashish Kolluri
School of Computing, NUS
Singapore

Ilya Sergey
University College London
United Kingdom

eek Saxena

Aquinas Hobor

Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts

SHELLY GROSSMAN, Tel Aviv University
ITTAI ABRAHAM, VMware Research
GUY GOLAN-GUETA, VMware Research
YAN MICHALEVSKY, Stanford University
NOAM RINETZKY, Tel Aviv University
MOOLY SAGIV, Tel Aviv University and VMware Research
YONI ZOHAR, Tel Aviv University

Automatic Generation of Precise and Useful Commutativity Conditions (Extended Version)

Kshitij Bansal[1*], Eric Koskinen[2†], and Omer Tripp[1‡]

[1] Google, Inc.
[2] Stevens Institute of Technology

Can we avoid those with
better Programming Language design?

The Goal of PL Design for Smart Contracts

Facilitate Reasoning about
High-Level Behaviour of Contracts
(as of Concurrent Objects)

Chapter 2

The Prototype

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)*

Automata2.v

State Context Goal Retract Undo Next Use Goto Qed Home Find Info Command Prooftree Interrupt Restart Help

```
Record Protocol (S : Type) :=
  CProt {
      (*Account id *)
      acc : address;
      (* Initial balance *)
      init_bal : ℕ;
      (* Initial state of a protocol *)
      init_state : S;
      (* Protocol comes with a set of transitions *)
      transitions : seq (ctransition S);
      (* All transitions have unique tags *)
      _ : uniq (map (@ttag _) transitions)
    }.

Definition tags {S : Type} (p : Protocol S) :=
  map (@ttag _) (transitions p).
End Protocol.

Section Semantics.
Variables (S : Type) (p : Protocol S).

(* Blockchain schedules *)
Definition schedule := seq (bstate * message).

(* In a well-formed schedule block numbers only grow *)
Fixpoint wf_sched (sch : schedule) :=
  if sch is s :: sch'
  then let bnum := block_num s.1 in
       all [pred s' | bnum ≤ block_num s'.1] sch' && wf_sched sch'
  else true.

Record step :=
  Step {
      pre  : cstate S;
      post : cstate S;
      out  : option message
  }.

Definition trace := seq step.
```

-:**- Automata2.v 24% (102,8) Git-master (Coq Script(0-) company- yas hs Outl Holes company)

Click (or press RET on) this bullet to hide or show its body.

U:--- *goals* All (1,0) (Coq Goals company)

step is defined
pre is defined
post is defined
out is defined

U:%%- *response* All (1,0) (Coq Response company WordWrap)

Automata2.v

State Context Goal Retract Undo Next Use Goto Qed Home Find Info Command Prooftree Interrupt Restart Help

```
Record Protocol (S : Type) :=
  CProt {
      (*Account id *)
      acc : address;
      (* Initial balance *)
      init_bal : ℕ;
      (* Initial state of a protocol *)
      init_state : S;
      (* Protocol comes with a set of transitions *)
      transitions : seq (ctransition S);
      (* All transitions have unique tags *)
      _ : uniq (map (@ttag _) transitions)
    }.

Definition tags {S : Type} (p : Protocol S) :=
  map (@ttag _) (transitions p).
End Protocol.

Section Semantics.
Variables (S : Type) (p : Protocol S).

(* Blockchain schedules *)
Definition schedule := seq (bstate * message).

(* In a well-formed schedule block numbers only grow *)
Fixpoint wf_sched (sch : schedule) :=
  if sch is s :: sch'
  then let bnum := block_num s.1 in
       all [pred s' | bnum ≤ block_num s'.1] sch' && wf_sched sch'
  else true.

Record step :=
  Step {
      pre  : cstate S;
      post : cstate S;
      out  : option message
  }.

Definition trace := seq step.
```

-:**- Automata2.v 24% (102,8) Git-master (Coq Script(0-) company- yas hs Outl Holes company)

Click (or press RET on) this bullet to hide or show its body.

U:--- *goals* All (1,0) (Coq Goals company)

```
step is defined
pre is defined
post is defined
out is defined
```

U:%%- *response* All (1,0) (Coq Response company WordWrap)

Automata2.v

State Context Goal Retract Undo Next Use Goto Qed Home Find Info Command Prooftree Interrupt Restart Help

```
Record Protocol (S : Type) :=
  CProt {
      (*Account id *)
      acc : address;
      (* Initial balance *)
      init_bal : N;
      (* Initial state of a protocol *)
      init_state : S;
      (* Protocol comes with a set of transitions *)
      transitions : seq (ctransition S);
      (* All transitions have unique tags *)
      _ : uniq (map (@ttag _) transitions)
    }.

Definition tags {S : Type} (p : Protocol S) :=
  map (@ttag _) (transitions p).
End Protocol.

Section Semantics.
Variables (S : Type) (p : Protocol S).

(* Blockchain schedules *)
Definition schedule := seq (bstate * message).

(* In a well-formed schedule block numbers only grow *)
Fixpoint wf_sched (sch : schedule) :=
  if sch is s :: sch'
  then let bnum := block_num s.1 in
       all [pred s' | bnum ≤ block_num s'.1] sch' && wf_sched sch'
  else true.

Record step :=
  Step {
      pre  : cstate S;
      post : cstate S;
      out  : option message
  }.

Definition trace := seq step.
```

-:**- Automata2.v 24% (102,8) Git-master (Coq Script(0-) company- yas hs Outl Holes company)

Click (or press RET on) this bullet to hide or show its body.

U:--- *goals* All (1,0) (Coq Goals company)

```
step is defined
pre is defined
post is defined
out is defined
```

U:%%- *response* All (1,0) (Coq Response company WordWrap)

Automata2.v

State Context Goal Retract Undo Next Use Goto Qed Home Find Info Command Prooftree Interrupt Restart Help

```
Record Protocol (S : Type) :=
  CProt {
      (*Account id *)
      acc : address;
      (* Initial balance *)
      init_bal : ℕ;
      (* Initial state of a protocol *)
      init_state : S;
      (* Protocol comes with a set of transitions *)
      transitions : seq (ctransition S);
      (* All transitions have unique tags *)
      _ : uniq (map (@ttag _) transitions)
    }.

Definition tags {S : Type} (p : Protocol S) :=
  map (@ttag _) (transitions p).
End Protocol.

Section Semantics.
Variables (S : Type) (p : Protocol S).

(* Blockchain schedules *)
Definition schedule := seq (bstate * message).

(* In a well-formed schedule block numbers only grow *)
Fixpoint wf_sched (sch : schedule) :=
  if sch is s :: sch'
  then let bnum := block_num s.1 in
         all [pred s' | bnum ≤ block_num s'.1] sch' && wf_sched sch'
  else true.

Record step :=
  Step {
      pre  : cstate S;
      post : cstate S;
      out  : option message
  }.

Definition trace := seq step.
```

-:**- Automata2.v 24% (102,8) Git-master (Coq Script(0-) company- yas hs Outl Holes company)

Click (or press RET on) this bullet to hide or show its body.

U:--- *goals* All (1,0) (Coq Goals company)

```
step is defined
pre is defined
post is defined
out is defined
```

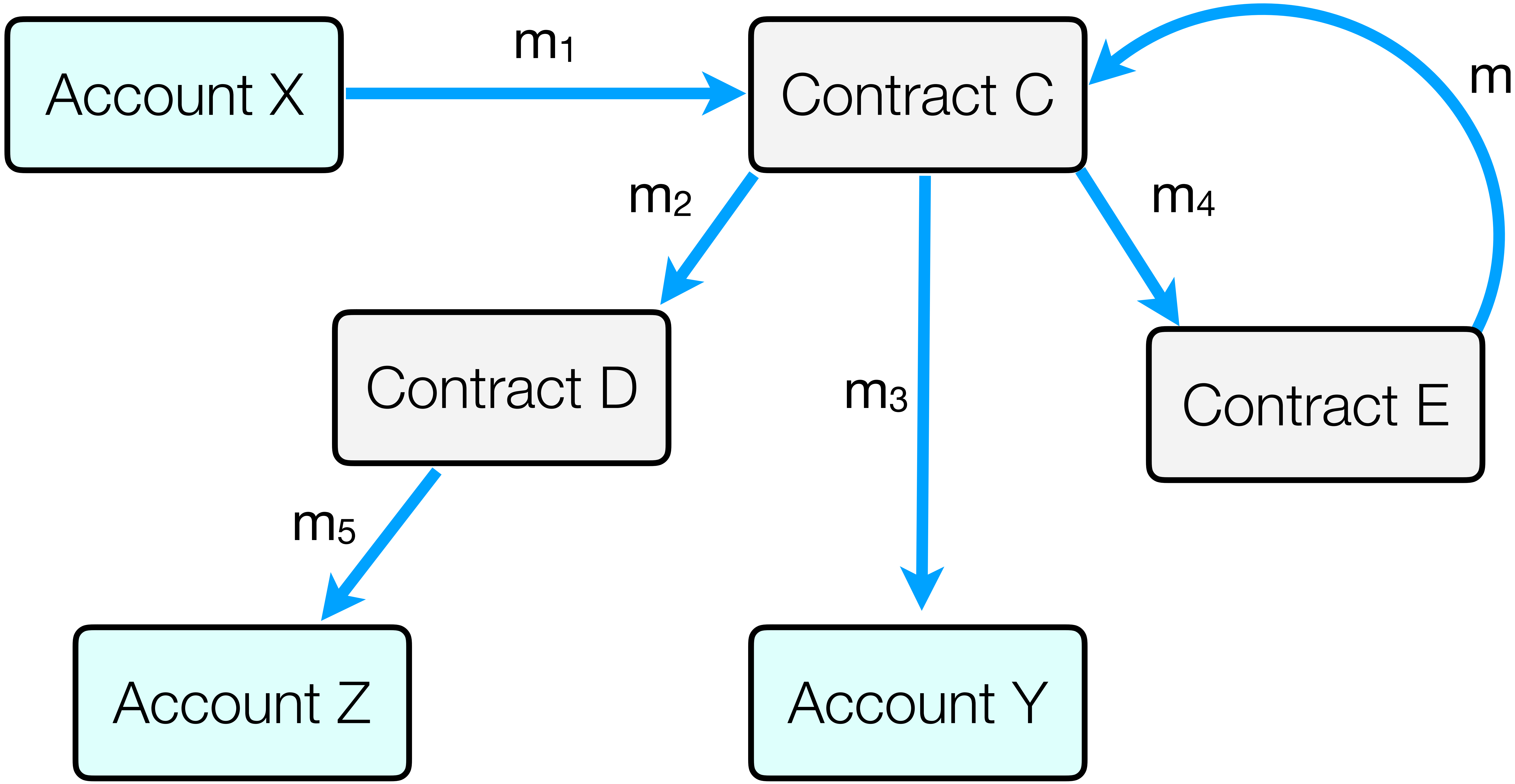
U:%%- *response* All (1,0) (Coq Response company WordWrap)

The Model

- Contracts are (infinite) *State-Transition Systems*
- Interaction *between* contracts via sending/receiving *messages*
- Messages trigger (effectful) *transitions* (sequences of *statements*)
- Most computations are done via *pure expressions*
- Contract's state is immutable parameters, mutable fields, balance

Contract Execution Model

Contract Execution Model



Contract Execution Model

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

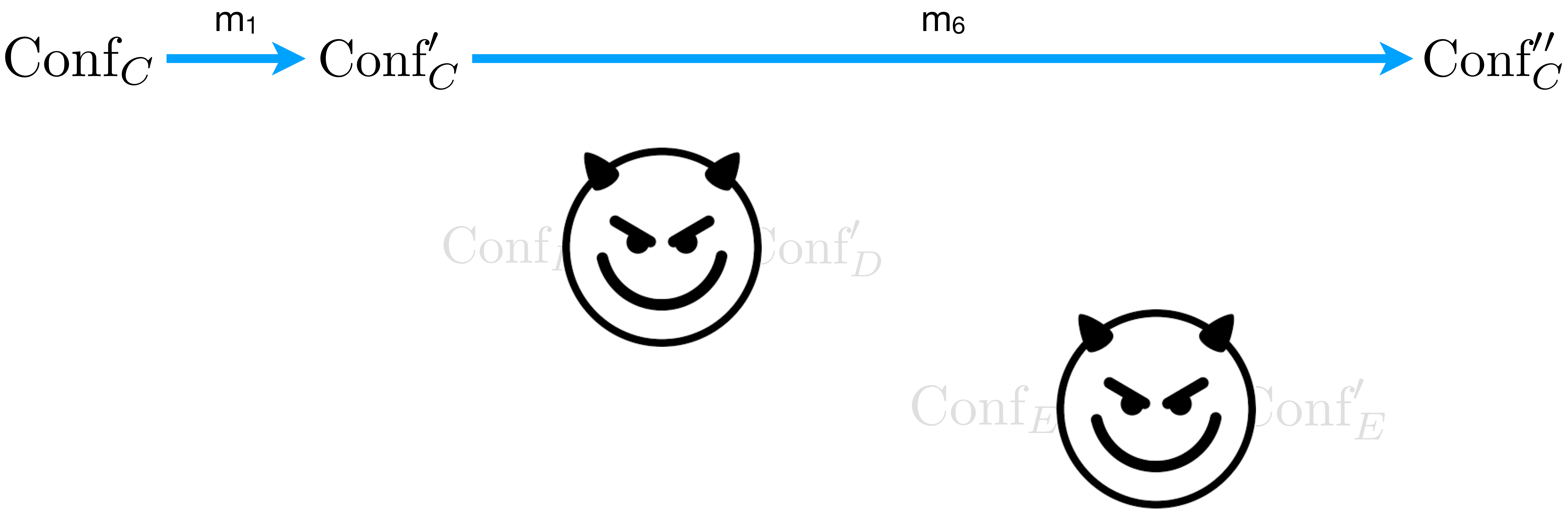
$$\mathrm{Conf}_D \xrightarrow{m_2} \boxed{\mathrm{Conf}'_D}$$

Final contract states

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

Fixed MAX length of call sequence

Contract Execution Model



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)

Temporal Properties

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

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

P holds here

R holds for intermediate messages

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

Chapter 3

The Proposal

A Secure Sharding Protocol For Open Blockchains

Loi Luu
National University of Singapore
loiluu@comp.nus.edu.sg

Viswesh Narayanan
National University of Singapore
visweshn@comp.nus.edu.sg

Chaodong Zheng
National University of Singapore
chaodong.zheng@comp.nus.edu.sg

Kunal Baweja
National University of Singapore
bawejaku@comp.nus.edu.sg

Seth Gilbert
National University of Singapore
seth.gilbert@comp.nus.edu.sg

Prateek Saxena
National University of Singapore
prateeks@comp.nus.edu.sg

A Secure Sharding Protocol For Open Blockchains

Loi Luu
National University of Singapore
loiluu@comp.nus.edu.sg

Viswesh Narayanan
National University of Singapore
visweshn@comp.nus.edu.sg

Chaodong Zheng
National University of Singapore
chaodong.zheng@comp.nus.edu.sg

Kunal Baweja
National University of Singapore
bawejaku@comp.nus.edu.sg

Seth Gilbert
National University of Singapore
seth.gilbert@comp.nus.edu.sg

Prateek Saxena
National University of Singapore
prateeks@comp.nus.edu.sg

The ZILLIQA Technical Whitepaper

[Version 0.1]

August 10, 2017

The ZILLIQA Team

www.zilliqa.com enquiry@zilliqa.com @zilliqa

zilliqa

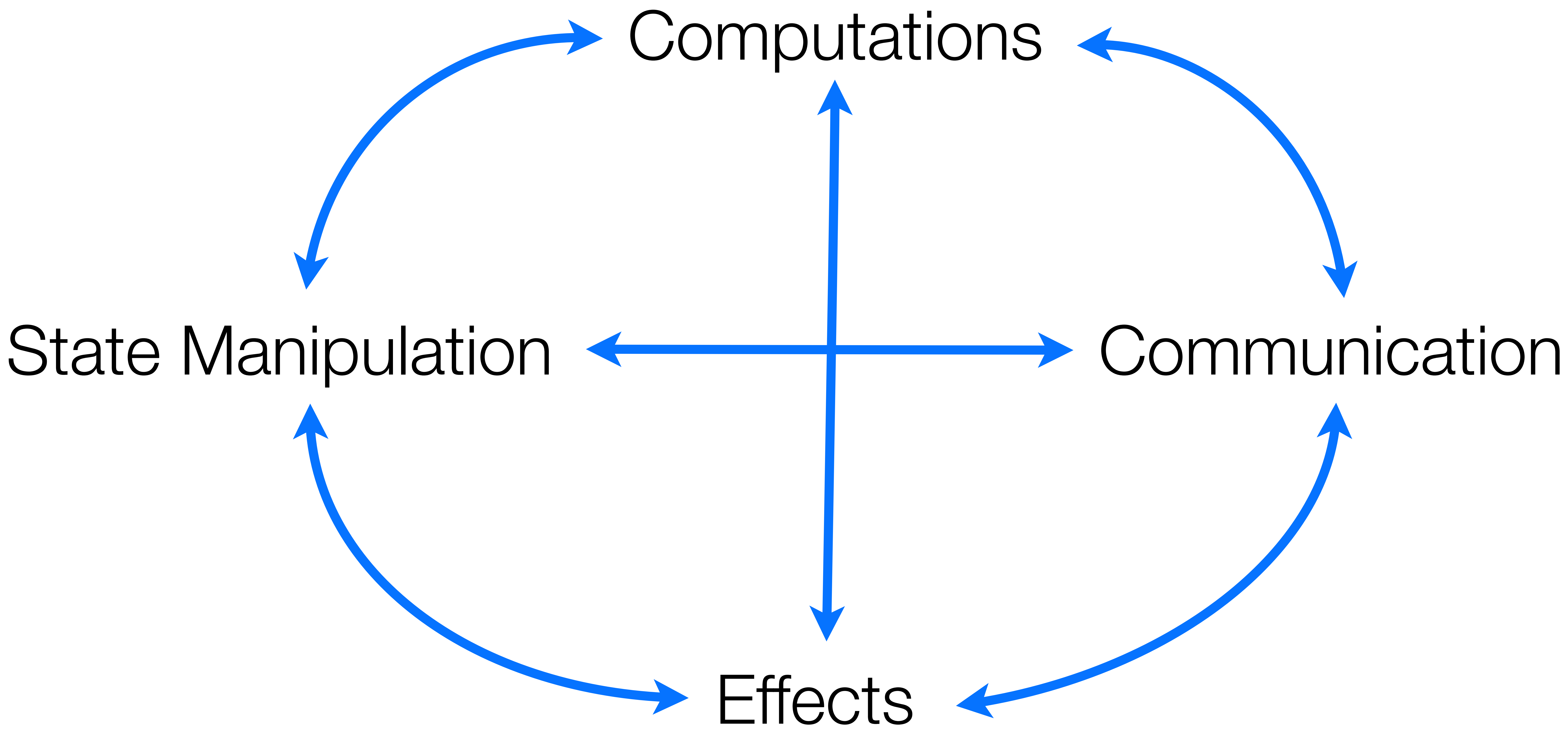

We’re building this cool sharded blockchain.
Would you like to help us a language for provably safe smart contracts?
Sure, I’d love to!
In fact, I might already have what you need…
… except you cannot really *run* in it yet.

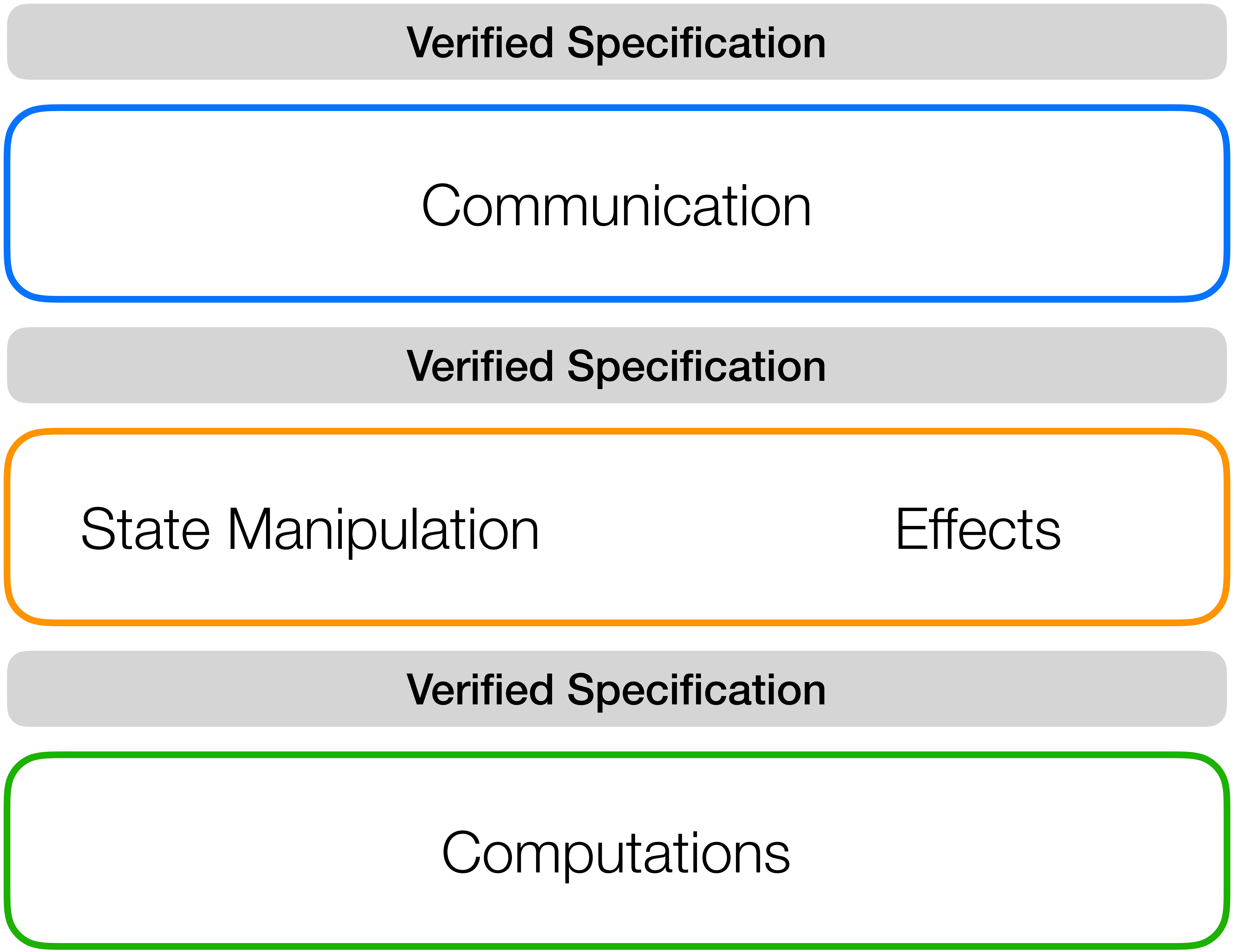
The Wish-List

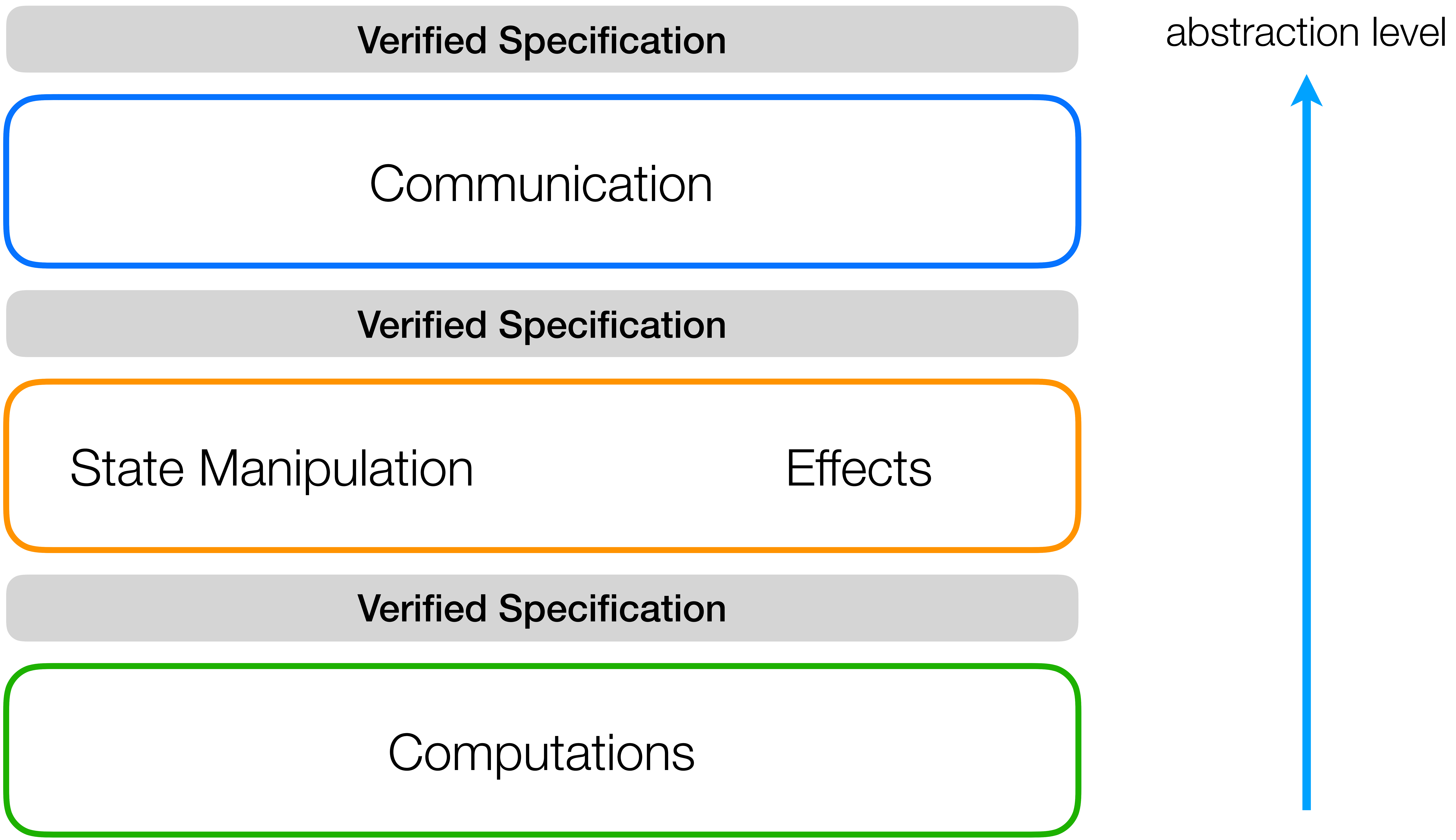
- Safety: basic fault avoidance checked ensured deployment
- Minimalism: simple to formalise and maintain
- Expressiveness: possible to implement common idioms
- Verification friendliness: tractable for automated and mechanised reasoning
- Performance: should not slow down the system’s throughput

The Essence of Smart Contracts

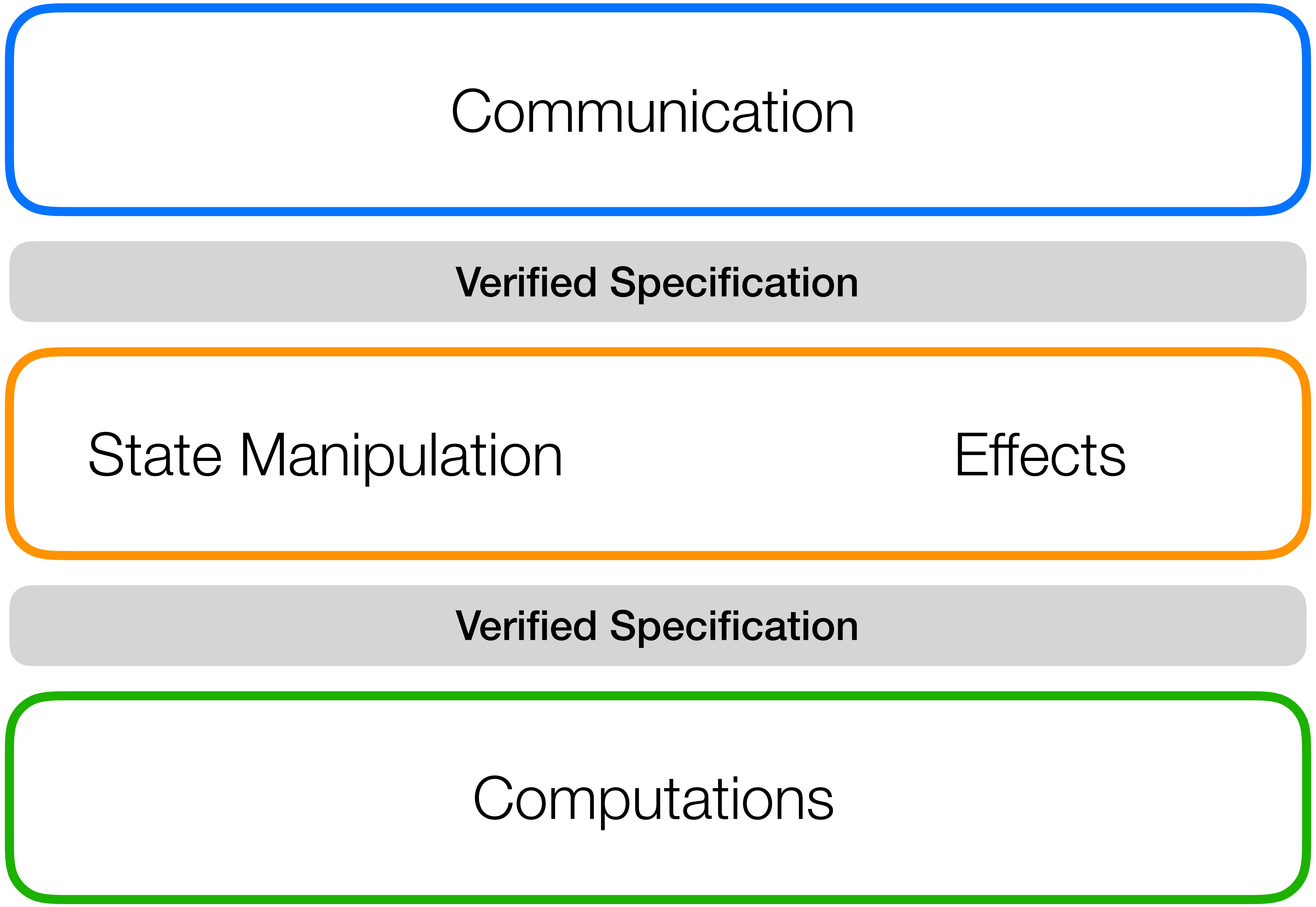
Simple Computations	self-explanatory
State Manipulation	changing contract's fields
Effects	accepting funds, logging events
Communication	sending funds, calling other contracts


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: a Smart Contract Intermediate-Level LAnguage

Automata for Smart Contract Implementation and Verification

Ilya Sergey
University College London
i.sergey@ucl.ac.uk

Amrit Kumar
National University of Singapore
amrit@comp.nus.edu.sg

Aquinas Hobor
Yale-NUS College
National University of Singapore
hobor@comp.nus.edu.sg

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

$$
\begin{array}{llcl}
\text{(signed integers)} & \mathit{int} & ::= & \text{i32} \mid \text{i64} \mid \text{i128} \mid \text{i256} \\
\text{(unsigned integers)} & \mathit{uint} & ::= & \text{u32} \mid \text{u64} \mid \text{u128} \mid \text{u256} \\
\text{(byte strings)} & \mathit{bst} & ::= & \text{bystrx } n \mid \text{bystr} \\
\text{(primitive types)} & \mathit{pt} & ::= & \mathit{int} \mid \mathit{uint} \mid \mathit{bst} \mid \\
 & & & \text{string} \mid \text{bnum} \mid \text{msg} \\
\text{(algebraic types)} & \mathcal{D} & ::= & \text{unit} \mid \text{bool} \mid \text{nat} \mid \text{option} \mid \\
 & & & \text{pair} \mid \text{list} \mid \text{U} \\
\text{(general Types)} & t & ::= & \mathit{pt} \mid \text{map } t\ t \mid t \rightarrow t \mid \\
 & & & \mathcal{D}\ \bar{t} \mid \alpha \mid \text{forall } \alpha .\ t
\end{array}
$$

Expressions (pure)

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

Structural Recursion with Lists

```
list_foldr: ∀ ‘A ‘B. ('A -> 'B -> 'B) -> 'B -> (List 'A) -> 'B
```

Element type

Result type

Iterator for non-empty list

Value for Nil

argument list

result

More Structural Recursion with Lists

“postponed” recursive call

```
list_foldk: ∀ ‘A ‘B. ('B -> 'A -> ('B -> 'B) -> 'B) ->
                    'B -> (List 'A) -> 'B
```

More Structural Recursion with Lists

```
let list_find : forall 'A. ('A -> Bool) -> List 'A -> Option 'A =
  tfun 'A =>
  fun (p : 'A -> Bool) =>
    let foldk = @list_foldk 'A (Option 'A) in
    let init = None {'A} in
    (* continue fold on None, exit fold when Some compare st. p(compare) *)
    let predicate_step =
      fun (ignore : Option 'A) => fun (x : 'A) =>
      fun (recurse: Option 'A -> Option 'A) =>
        let p_x = p x in
        match p_x with
        | True => Some {'A} x
        | False => recurse init
        end in
    foldk predicate_step init
```

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
	`event` `m`	create a single event
	`send` `ms`	send list of messages
	`throw`	abort the execution
	`in-place map operations`	efficient manipulation with maps

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

```
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
```

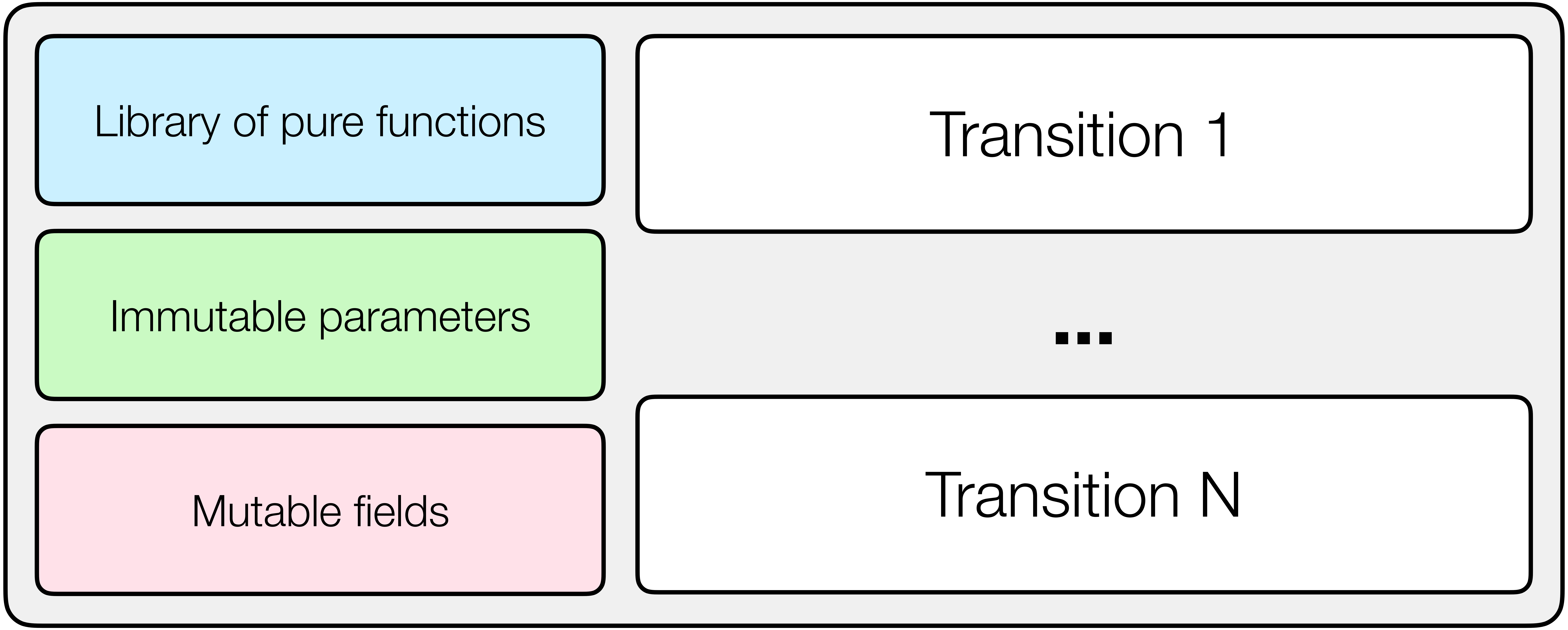
```
transition Donate (sender: Address, amount: Int)
  blk <- & BLOCKNUMBER;
  in_time = blk_leq blk max_block;
  match in_time with
  | True  =>
    bs  <- backers;
    res = check_update bs sender amount;
    match res with
    | None =>
      msg  = {tag : Main; to : sender; amount : 0; code : already_backed};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

Contract Structure



On-Chain Deployment

- Scilla contracts are *interpreted* (not compiled before deployment)
- A contract *cannot* explicitly refer to another contract’s state
- However, pure *libraries* can be freely reused
- One may deploy *a* library even *without* a contract

Gas Accounting

- **Simple term reductions**: 1
- **Pattern matching**: (size of patterns) * (number of branches)
- **Built-in operations**: proportional to the size of arguments
- **Map manipulations**: proportional to the size of maps
- Also charging *parser* and the *type-checker* (run by miners)

Scilla Interpreter

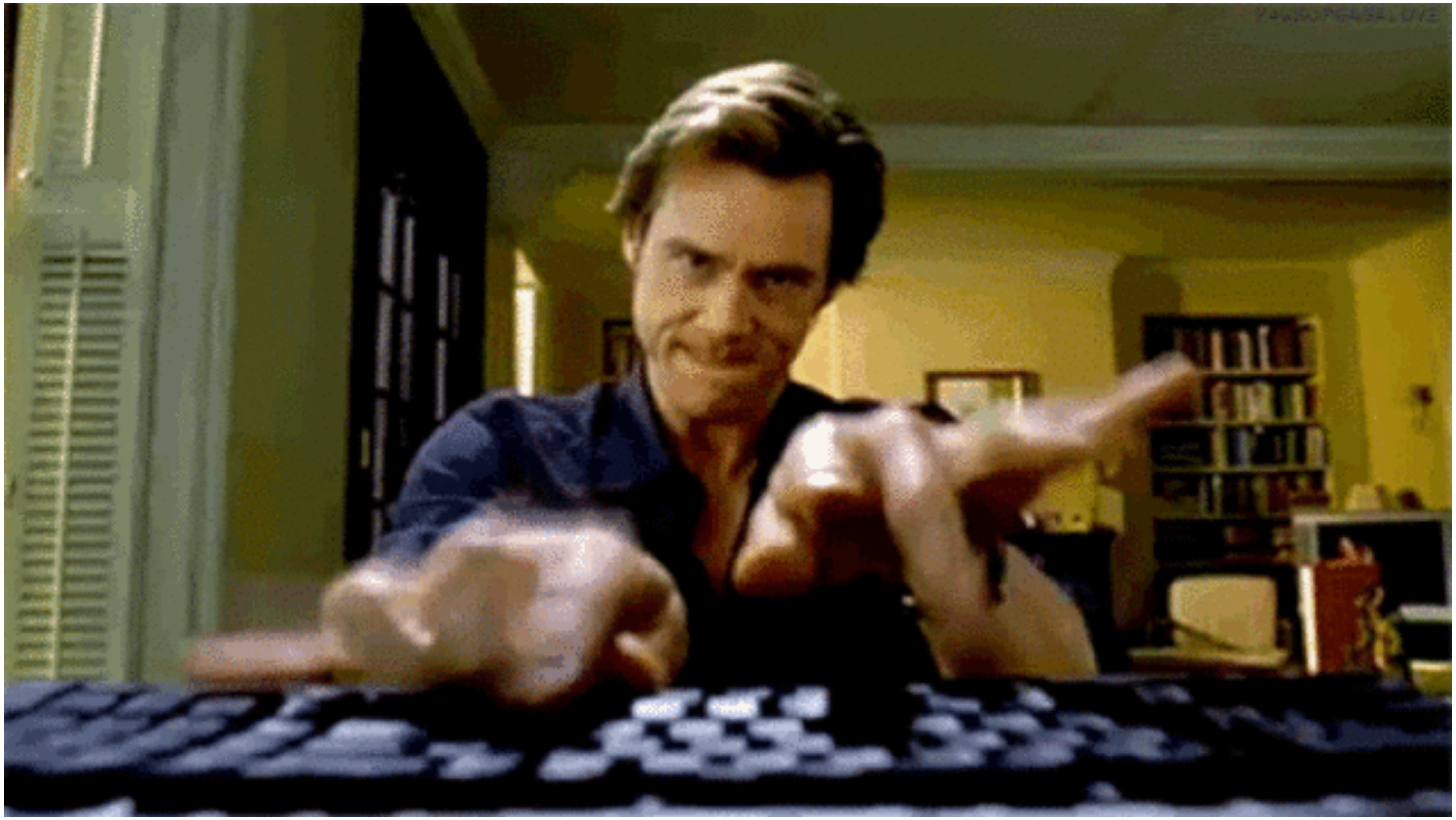
```
(***********************************************************)
(* A monadic big-step evaluator for Scilla expressions *)
(***********************************************************)

(* [Evaluation in CPS]

   The following evaluator is implemented in a monadic style, with the
   monad, at the moment to be CPS, with the specialised return result
   type as described in [Specialising the Return Type of Closures].
 *)

let rec exp_eval erep env  =
  let (e, loc) = erep in
  match e with
  | Literal l ->
      pure (l, env)
  | Var i ->
      let%bind v = Env.lookup env i in
      pure @@ (v, env)
  | Let (i, _, lhs, rhs) ->
      let%bind (lval, _) = exp_eval_wrapper lhs env in
      let env' = Env.bind env (get_id i) lval in
      exp_eval_wrapper rhs env'
  | Message bs ->
```

- Core: about 200 LOC of OCaml
- Monadic style:
  error handling, gas accounting, continuation passing
- Changes in gas accounting *have not affected* the core interpreter
- Lots of performance bottlenecks fixed *without ever touching* the evaluator (CPS refactoring)

Chapter 4

The Evaluation

Expressivity



Expressivity

- Standard Library: ~1 kLOC

Contract	LOC	#Lib	#Trans
HelloWorld	31	3	2
Crowdfunding	127	13	3
Auction	140	11	3
ERC20	158	2	6
ERC721	270	15	6
Wallet	363	28	9
Bookstore	123	6	3
HashGame	209	16	3
Schnorr	71	2	3

Verification-Friendliness

- A framework for staged static analyses (optional)
- Two instances:
  - Gas-Usage Analysis
  - Cash-Flow Analysis

Gas Usage Analysis

- Soundly derives a *gas usage polynomial*
- *Folds* allow for *simple recurrences*, solved statically
- *Compositional*, GU signatures are cached

Modular, Higher-Order Cardinality Analysis in Theory and Practice

Ilya Sergey
IMDEA Software Institute
ilya.sergey@imdea.org

Dimitrios Vytiniotis   Simon Peyton Jones
Microsoft Research
{dimitris,simonpj}@microsoft.com

Gas Usage Analysis

```
(* forall 'A. forall 'B. ('A → 'B) → List 'A → List 'B *)
let list_map = tfun 'A ⇒ tfun 'B ⇒
  fun (f : 'A → 'B) ⇒ fun (l : List 'A) ⇒
  let folder = @list_foldr 'A (List 'B) in
  let init = Nil {'B} in
  let iter = fun (h : 'A) ⇒ fun (z : List 'B) ⇒
    let h1 = f h in
    Cons {'B} h1 z
  in folder iter init l
```

```
Parameter list:  [f, l]
Gas consumption: 5(a) + 1(a)(b) + 11
Legend: a: Length of: l; b: Cost of calling f on (Element of: l)
```

Cash-Flow Analysis

- Soundly determines what fields *represent money*
- Takes use input for *custom tokens*
- Based on simple abstract interpretation

Lattice of Cash Tags

$$\begin{array}{rcl} \tau & ::= & \textbf{Money} \mid \textbf{NotMoney} \mid \textbf{Map}\ \tau \mid t\ \overline{\tau} \mid \top \mid \bot \\ t & ::= & \textbf{Option} \mid \textbf{Pair} \mid \textbf{List} \mid \ \ldots \end{array}$$

$$\begin{array}{lrcll} \text{(maps)} & \textbf{Map}\ \tau & \sqsubseteq & \textbf{Map}\ \tau' & \text{iff } \tau \sqsubseteq \tau' \\ \text{(algebraic types)} & t\ \overline{\tau} & \sqsubseteq & t'\ \overline{\tau'} & \text{iff } t = t' \text{ and } \tau_i \sqsubseteq \tau'_i \text{ for all } i \\ \text{(bottom)} & \bot & \sqsubseteq & \tau & \text{for all } \tau \\ \text{(top)} & \tau & \sqsubseteq & \top & \text{for all } \tau \end{array}$$

Results for Crowdfunding

Field/Param	Tag
`owner`	**NotMoney**
`max_block`	**NotMoney**
`goal`	**Money**
`backers`	**Map Money**
`funded`	**NotMoney**

Analysis Results

Contract	LOC	#Lib	#Trans
HelloWorld	31	3	2
Crowdfunding	127	13	3
Auction	140	11	3
ERC20	158	2	6
ERC721	270	15	6
Wallet	363	28	9
Bookstore	123	6	3
HashGame	209	16	3
Schnorr	71	2	3

Gas Usage Analysis

Contract	LOC	#Lib	#Trans	Asympt. GU
HelloWorld	31	3	2	$O(\lvert\text{string}\rvert)$
Crowdfunding	127	13	3	$O(\lvert\text{map}\rvert)$
Auction	140	11	3	$O(\lvert\text{map}\rvert)$
ERC20	158	2	6	$O(1)$
ERC721	270	15	6	$O(\lvert\text{map}\rvert)$
Wallet	363	28	9	$O(\lvert\text{map}\rvert \times \lvert\text{list}\rvert)$
Bookstore	123	6	3	$O(\lvert\text{string}\rvert + \lvert\text{map}\rvert)$
HashGame	209	16	3	$O(1)$
Schnorr	71	2	3	$O(\lvert\text{bystr}\rvert)$

Cash-Flow Analysis

Contract	LOC	#Lib	#Trans	Asympt. GU	\$-Flow
HelloWorld	31	3	2	$O(\|string\|)$	✓
Crowdfunding	127	13	3	$O(\|map\|)$	✓
Auction	140	11	3	$O(\|map\|)$	✓
ERC20	158	2	6	$O(1)$	$\checkmark^*$
ERC721	270	15	6	$O(\|map\|)$	$\checkmark_\perp$
Wallet	363	28	9	$O(\|map\| \times \|list\|)$	✓
Bookstore	123	6	3	$O(\|string\| + \|map\|)$	✓
HashGame	209	16	3	$O(1)$	✓
Schnorr	71	2	3	$O(\|bystr\|)$	✓

Cash-Flow Analysis

Contract	LOC	#Lib	#Trans	Asympt. GU	$-Flow
HelloWorld	31	3	2	$O(\lvert\text{string}\rvert)$	✓
Crowdfunding	127	13	3	$O(\lvert\text{map}\rvert)$	✓
Auction	140	11	3	$O(\lvert\text{map}\rvert)$	✓
ERC20	158	2	6	$O(1)$	✓*
ERC721	270	15	6	$O(\lvert\text{map}\rvert)$	$\checkmark_{\perp}$
Wallet	363	28	9	$O(\lvert\text{map}\rvert \times \lvert\text{list}\rvert)$	✓
Bookstore	123	6	3	$O(\lvert\text{string}\rvert + \lvert\text{map}\rvert)$	✓
HashGame	209	16	3	$O(1)$	✓
Schnorr	71	2	3	$O(\lvert\text{bystr}\rvert)$	✓

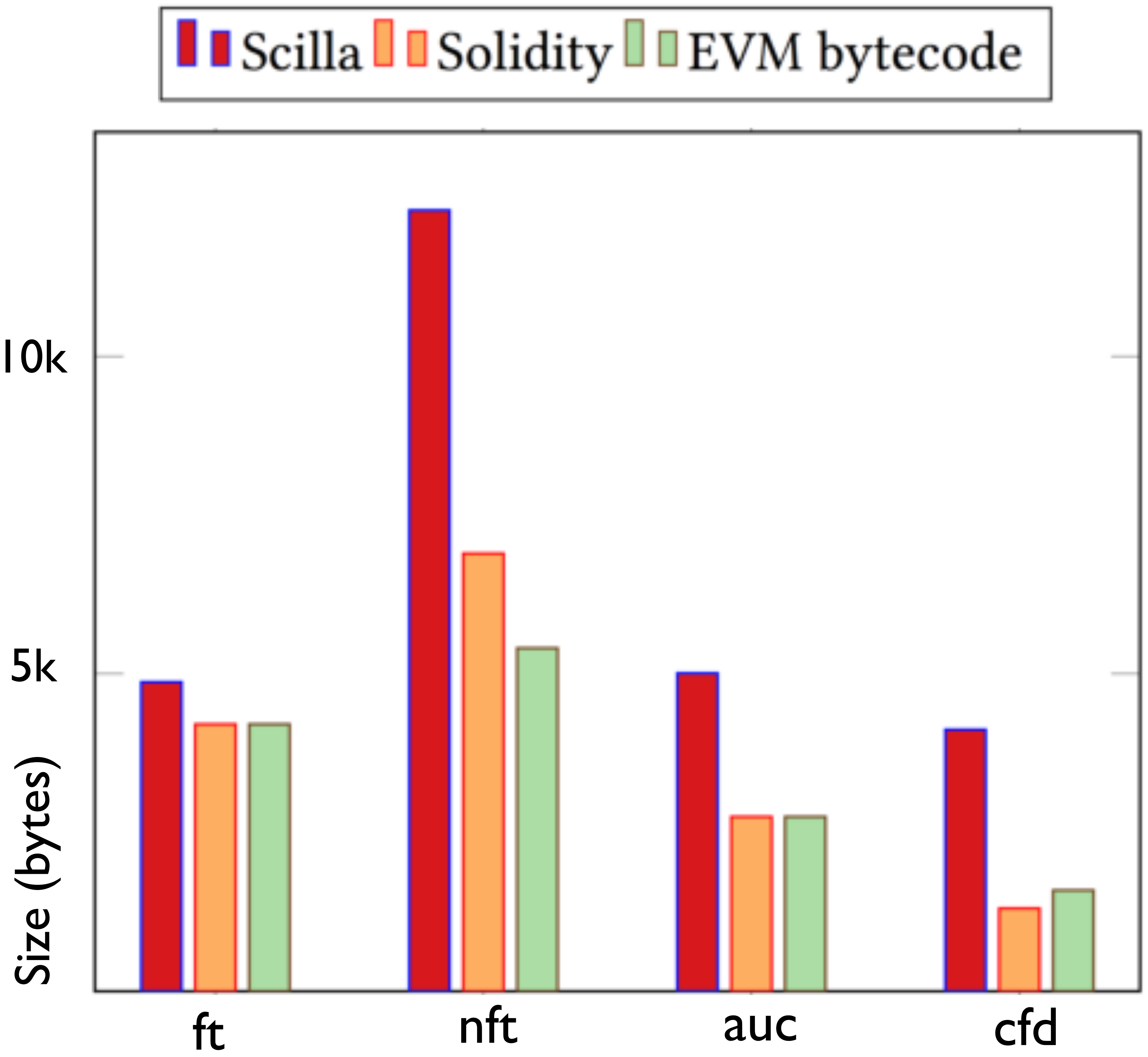
non-native tokens

Cash-Flow Analysis

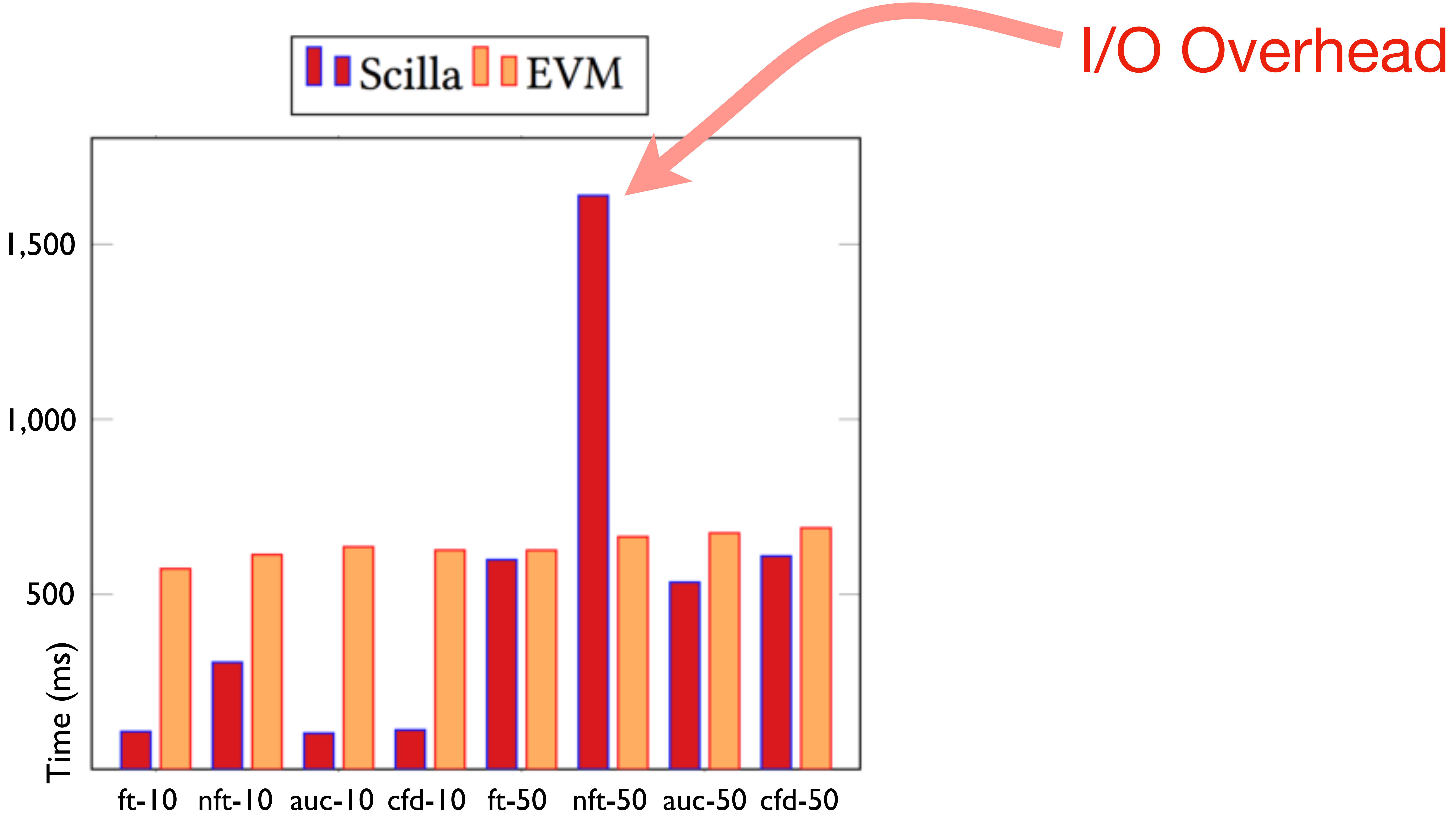
Contract	LOC	#Lib	#Trans	Asympt. GU	\$-Flow
HelloWorld	31	3	2	$O(\lvert\text{string}\rvert)$	✓
Crowdfunding	127	13	3	$O(\lvert\text{map}\rvert)$	✓
Auction	140	11	3	$O(\lvert\text{map}\rvert)$	✓
ERC20	158	2	6	$O(1)$	✓*
ERC721	270	15	6	$O(\lvert\text{map}\rvert)$	$\checkmark_{\perp}$
Wallet	363	28	9	$O(\lvert\text{map}\rvert \times \lvert\text{list}\rvert)$	✓
Bookstore	123	6	3	$O(\lvert\text{string}\rvert + \lvert\text{map}\rvert)$	✓
HashGame	209	16	3	$O(1)$	✓
Schnorr	71	2	3	$O(\lvert\text{bystr}\rvert)$	✓

non-fungible tokens

Relative Code Size



Performance



Chapter 5

The Challenges

Maps that Grow

```
(*************************************************)
(*              The contract definition         *)
(*************************************************)
contract Crowdfunding

(*  Parameters *)
(owner     : ByStr20,
 max_block : BNum,
 goal      : Uint128)

(* Mutable fields *)
field backers : Map ByStr20 Uint128 = Emp ByStr20 Uint128
field funded : Bool = False

transition Donate ()
  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 =>
      e = {_eventname : "DonationFailure"; donor : _sender; amount : _amount;
      event e
    | Some bs1 =>
      backers := bs1;
```

Who has donated

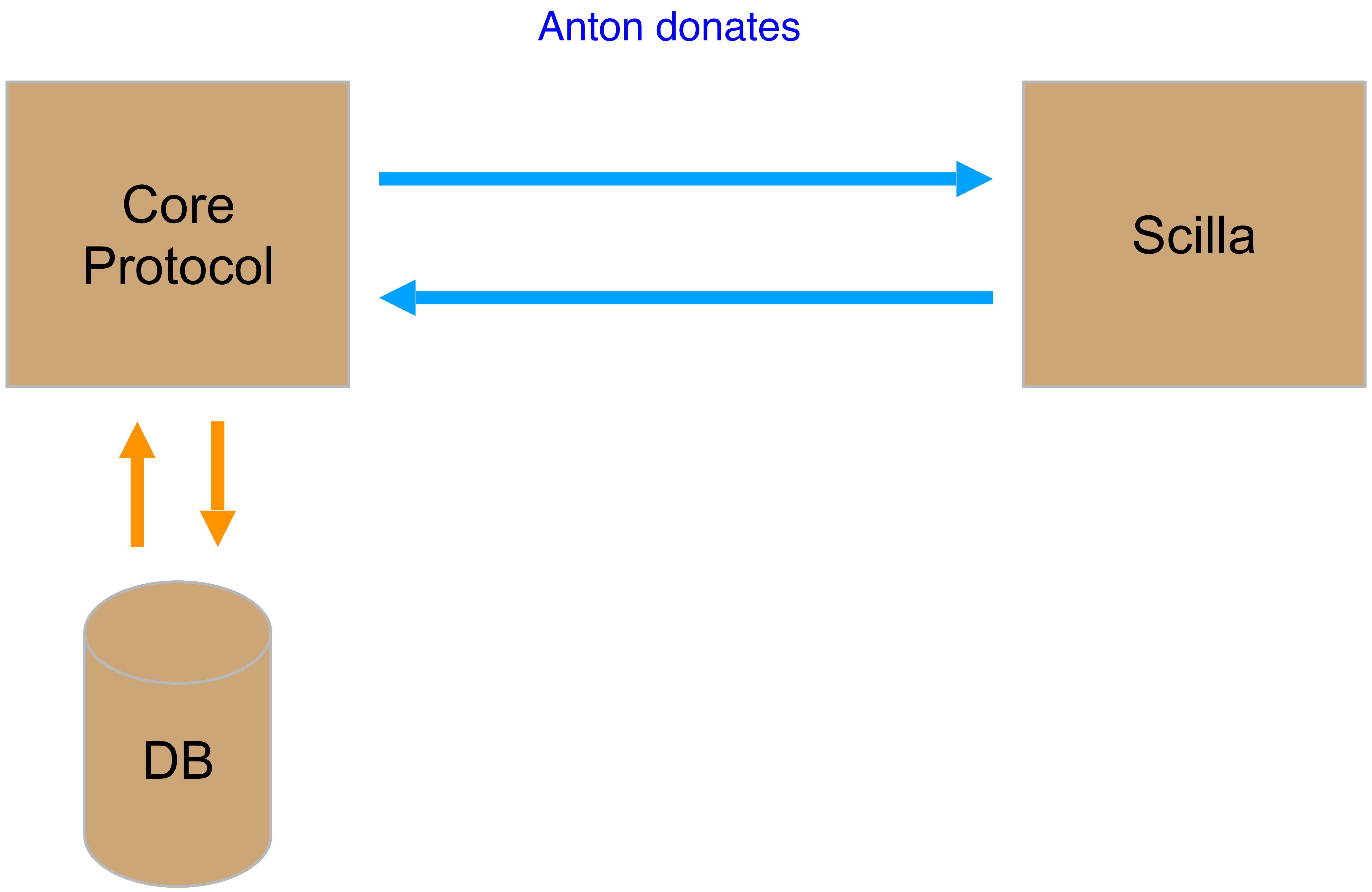
How much

Donor	Amount
Amrit	100
Jacob	120
Vaivas	500000

The table can grow very large!

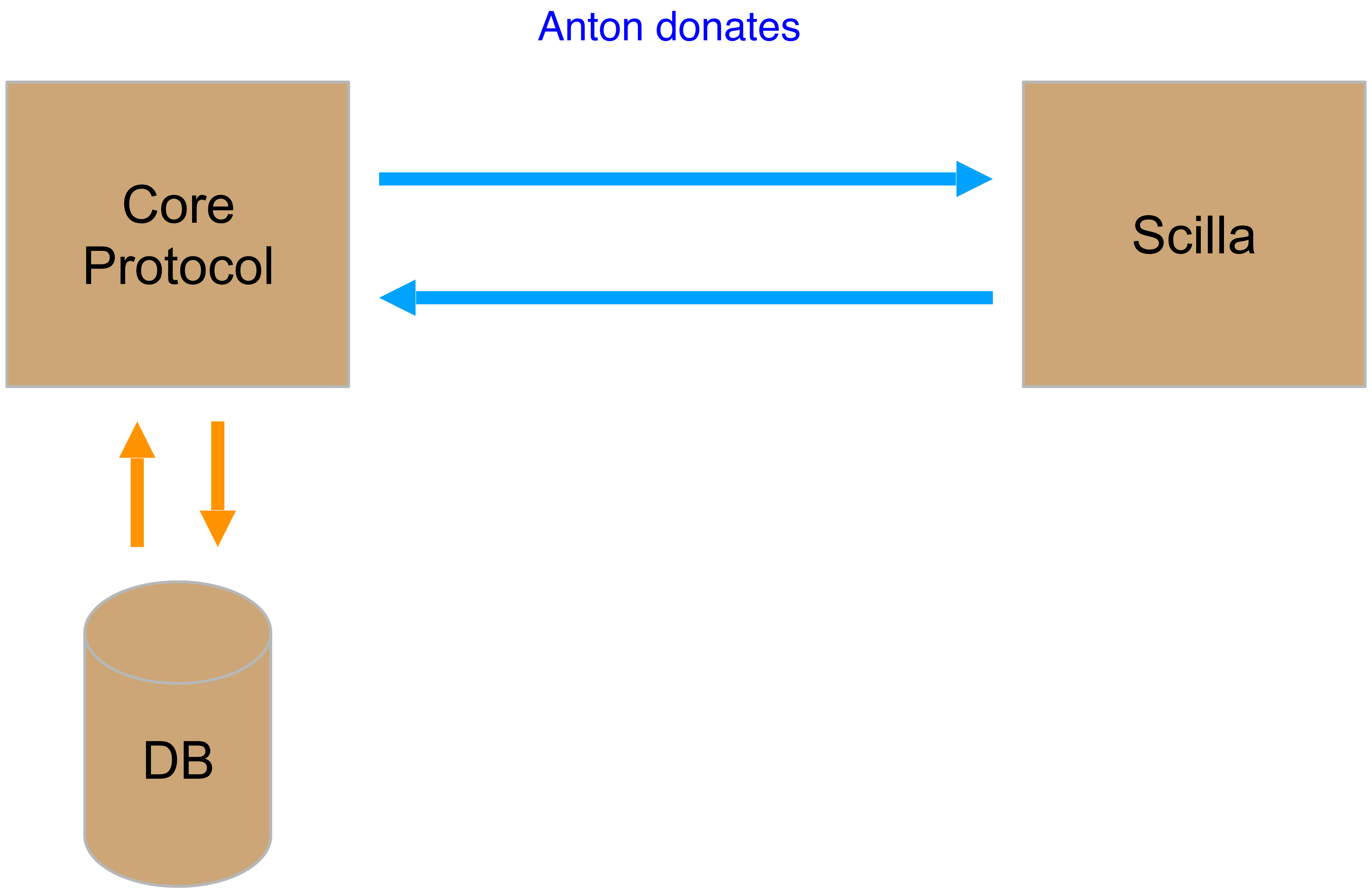
Initial Naïve Execution Model

Donor	Amount
Amrit	100
Jacob	120
Vaivas	500000

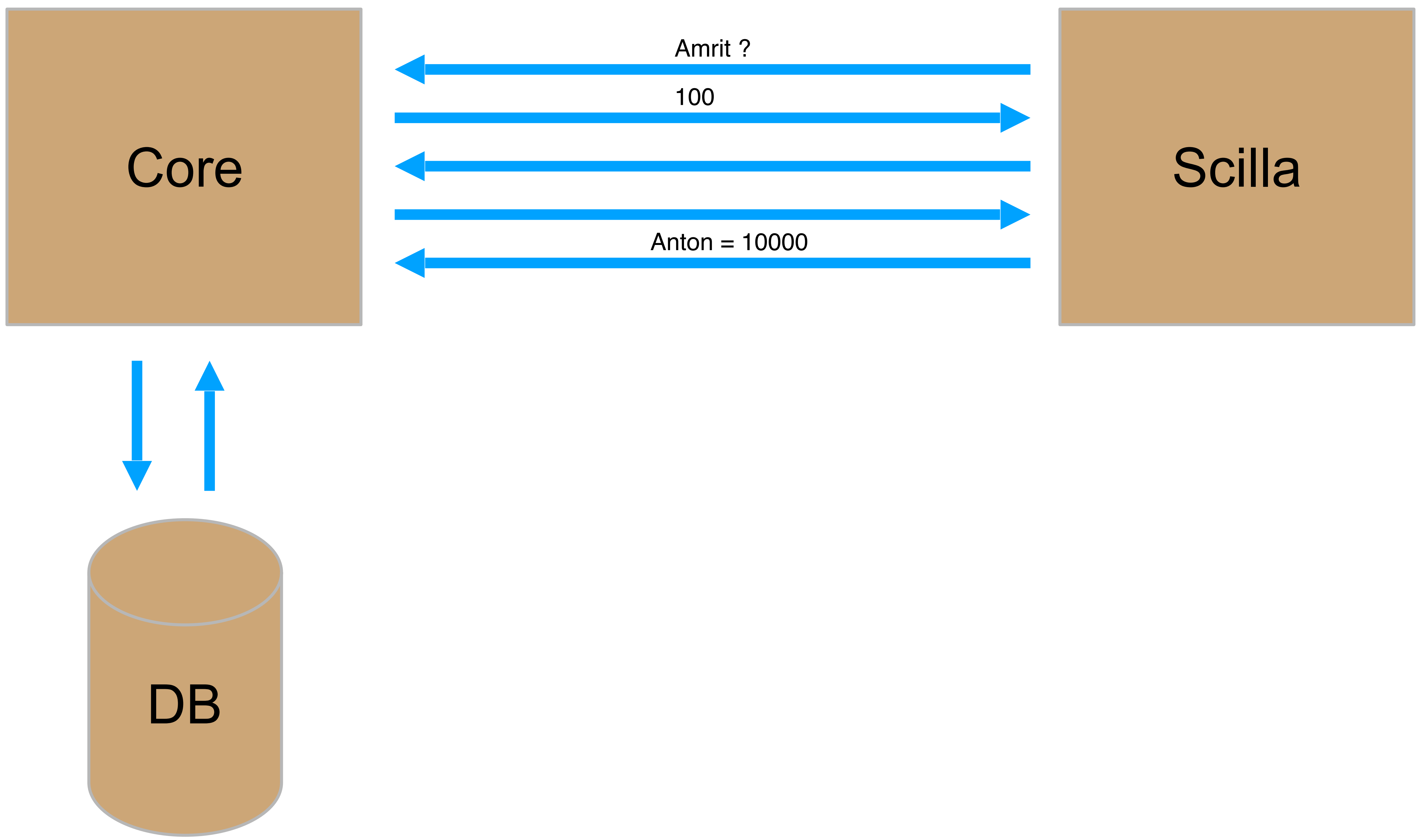


Initial Naïve Execution Model

Donor	Amount
Amrit	100
Jacob	120
Vaivas	500000
Anton	10000



Fine-Grained Interaction



The IPC Protocol

- Core blockchain distinguishes between **map** and **non-map** fields in a Scilla contract, *optimising* map key accesses upon deployment

- Still, *no changes* in the core interpreter

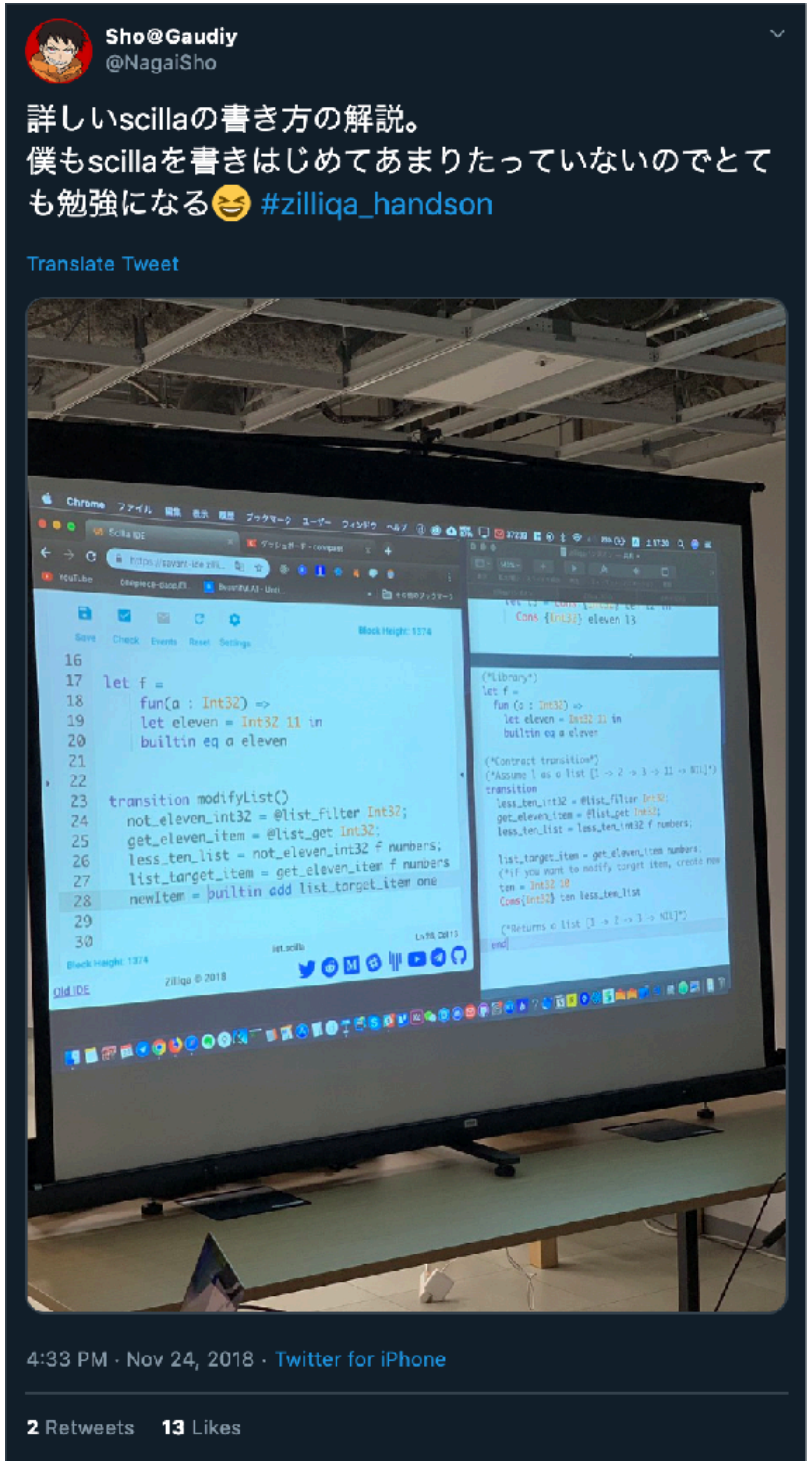
  - All change is encapsulated in the *Evaluator monad*

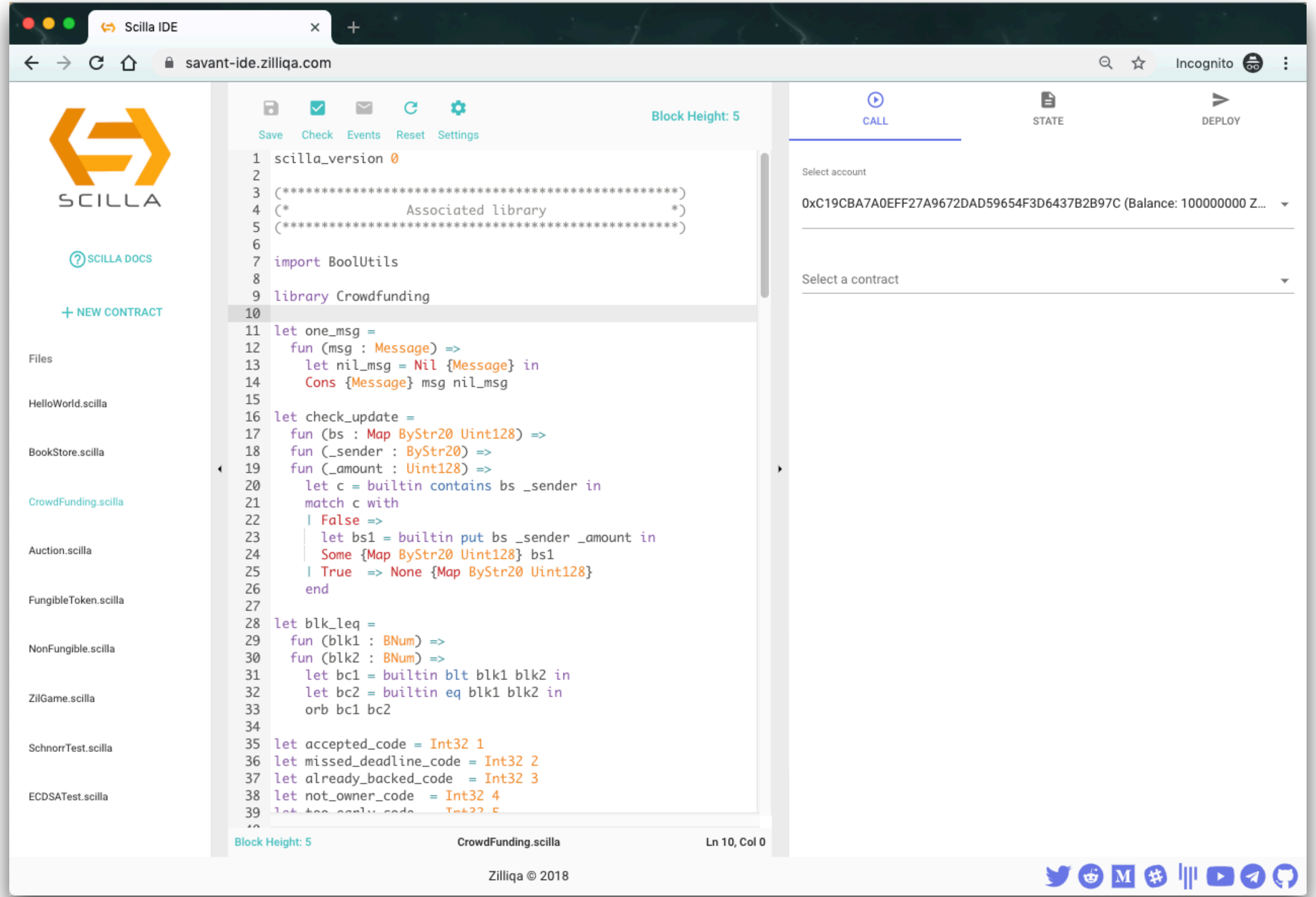
Chapter 6

The Big Picture

Adoption

- Scilla launched on Zilliqa mainnet since June 2019
- Dozens of community-contributed contracts:
  - ERC223, ERC777
  - contracts for crowdsales, escrows
  - contracts for access control
  - upcoming standard ERC1404 for security tokens
- Language-Server Protocol Support
- Emacs and VSCode plugins (w/ semantic highlighting)
- Workshops, tutorials, developer sessions





Scilla IDE
savant-ide.zilliqa.com
Incognito
SCILLA
SCILLA DOCS
+ NEW CONTRACT
Files
HelloWorld.scilla
BookStore.scilla
CrowdFunding.scilla
Auction.scilla
FungibleToken.scilla
NonFungible.scilla
ZilGame.scilla
SchnorrTest.scilla
ECDSATest.scilla
Save
Check
Events
Reset
Settings
Block Height: 5
scilla_version 0
(***)
(*   Associated library   *)
(***)
import BoolUtils
library Crowdfunding
let one_msg =
  fun (msg : Message) =>
    let nil_msg = Nil {Message} in
    Cons {Message} msg nil_msg
let check_update =
  fun (bs : Map ByStr20 Uint128) =>
  fun (_sender : ByStr20) =>
  fun (_amount : Uint128) =>
    let c = builtin contains bs _sender in
    match c with
    | False =>
      let bs1 = builtin put bs _sender _amount in
      Some {Map ByStr20 Uint128} bs1
    | True => None {Map ByStr20 Uint128}
    end
let blk_leq =
  fun (blk1 : BNum) =>
  fun (blk2 : BNum) =>
    let bc1 = builtin blt blk1 blk2 in
    let bc2 = builtin eq blk1 blk2 in
    orb bc1 bc2
let accepted_code = Int32 1
let missed_deadline_code = Int32 2
let already_backed_code = Int32 3
let not_owner_code = Int32 4
Block Height: 5
CrowdFunding.scilla
Ln 10, Col 0
CALL
STATE
DEPLOY
Select account
0xC19CBA7A0EFF27A9672DAD59654F3D6437B2B97C (Balance: 100000000 Z...
Select a contract
Zilliqa © 2018

ViewBlock | Zilliqa Contract zil

viewblock.io/zilliqa/address/zil1w0gj7tnxk8usu68et44jfchuwh0mjc040pqd6l?tab=code

Incognito

ZILLIQA

mainnet

Search for a tx, address, name or block.

Sign in

ADDRESSES TRANSACTIONS BLOCKS STATS API

Price	Market Cap	Volume
$0.006	$52.72M	$27.57M

Contract

zil1w0gj7tnxk8usu68et44jfchuwh0mjc040pqd6l

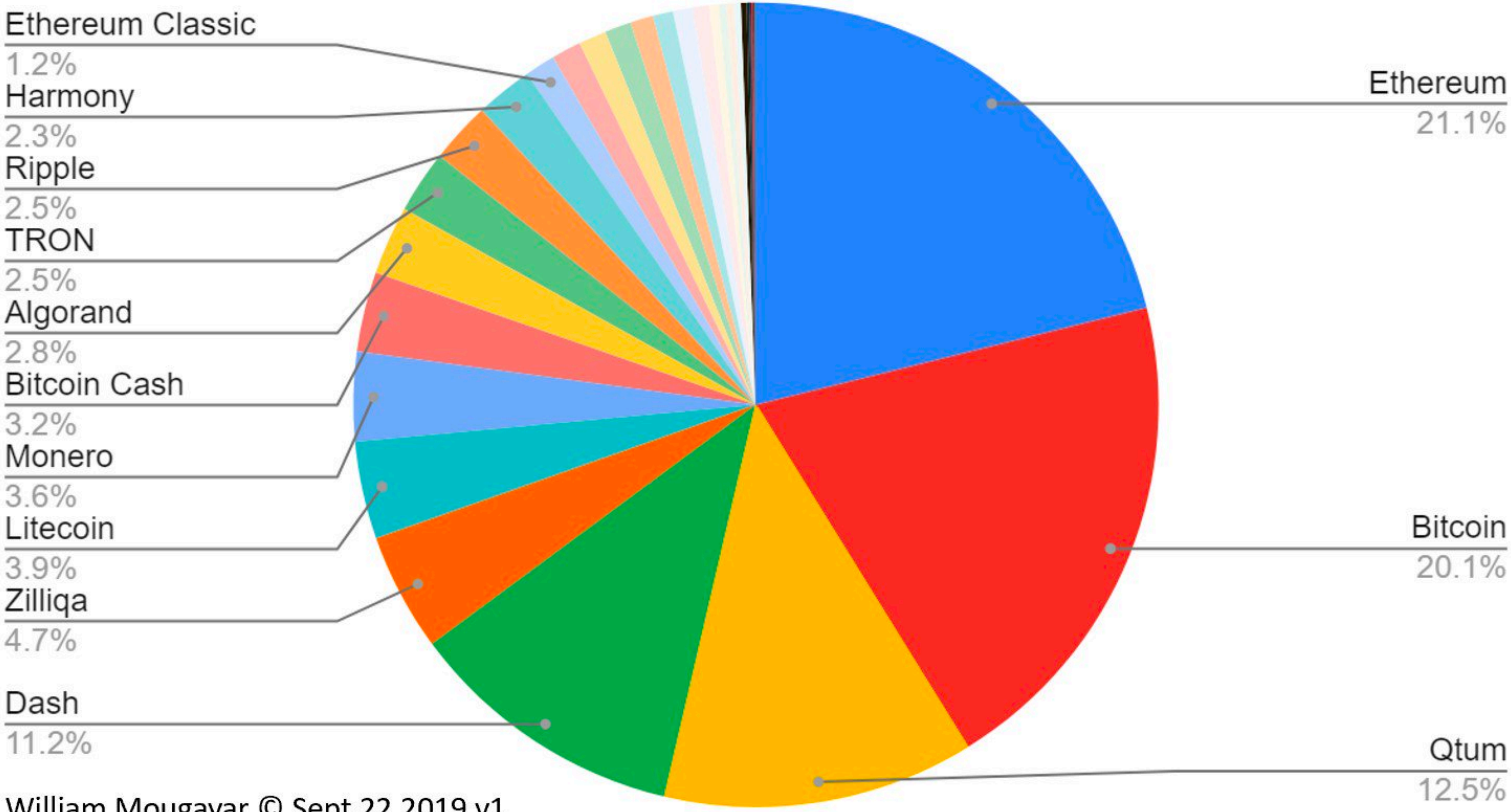
COPY ADDRESS QR

Balance	Transactions	Contract Creation
193.35 ZIL	40	zil1fxx... at dab4e4878c0d93abcfaa...

TRANSACTIONS CODE

```
scilla_version 0

import BoolUtils
library Exchange
let zero_address = 0x0000000000000000000000000000000000000000
let zero = Uint128 0

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

(* error codes library *)
let code_success = Uint32 0
```

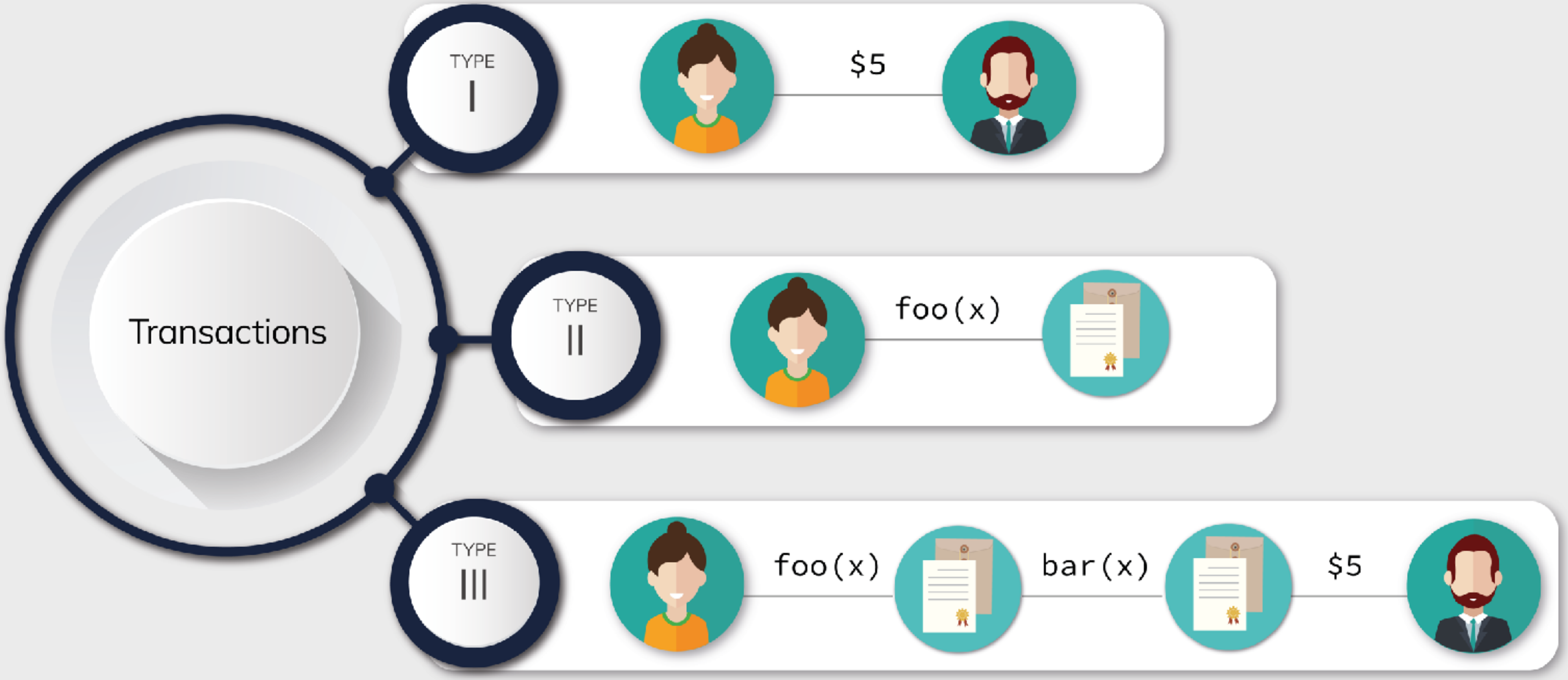
Global Ranks by # of Active (validating) Blockchain Nodes



William Mougayar © Sept 22 2019 v1

Scilla on a Sharded Blockchain

TYPES OF TRANSACTIONS



Chapter 7

The Future

Work in Progress

- *Full* Scilla to Coq translation (coming soon)
- Type-preserving compilation into an efficient back-end (LLVM)
- Certifications for *Proof-Carrying Code* (storable on a blockchain)
- More automated analyses

CHAINSECURITY

Epilogue

Lessons Learned

- Growing a new smart contract language is a rollercoaster of *excitement* and *angst.*
- Functional programming is a great way to keep the language *minimalistic* yet *expressive.*
- The language will be forced to *grow* and *change* — just embrace it.
- Yet, **lots of ideas from PL research** can be reused with *very low overhead* on implementation and adoption.
- It pays off to build an enthusiastic developer community: more feedback — more *informed design choices.*

Research Challenges

- Exploiting static properties of smart contracts for faster consensus
- Robust and adequate gas cost assignment
- Optimising compilers — good or evil?

SCILLA

http://scilla-lang.org

OOPSLA'19

Artifacts Available acm

Artifacts Evaluated Reusable acm

Artifacts Evaluated Functional acm

Safer Smart Contract Programming with SCILLA

ILYA SERGEY, Yale-NUS College, Singapore and National University of Singapore, Singapore

Research, India

h, Denmark

ted Kingdom

Russia

earch, Malaysia

Research Grants →

SCAN ME

Thanks!

CertiChain Project

- **Postdoc/PhD positions** on *formal proofs* for *distributed systems* and *smart contracts* at **Yale-NUS College** and **NUS School of Computing** are available **now**.