The Scilla Journey: From Proof General to Thousands of Nodes

Ilya Sergey

ilyasergey.net
Prologue

The Technology
Blockchain Consensus

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

\[ \{tx_1, tx_3, tx_5, tx_4, tx_2\} \]

transactions can be anything

blockchain consensus protocol

\[ tx_5 \rightarrow tx_3 \rightarrow tx_4 \rightarrow tx_1 \rightarrow tx_2 \]
Blockchain Consensus

\[ \{tx_1, tx_3, tx_5, tx_4, tx_2\} \]

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

\[ tx_5 \rightarrow tx_3 \rightarrow tx_4 \rightarrow tx_1 \rightarrow tx_2 \]
Blockchain Consensus

\[ \{tx_1, tx_3, tx_5, tx_4, tx_2\} \]

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

\[ tx_5 \rightarrow tx_3 \rightarrow tx_4 \rightarrow tx_1 \rightarrow tx_2 \]
Blockchain Consensus

\[
\{tx_1, tx_3, tx_5, tx_4, tx_2\}
\]

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

GB = genesis block

\[
tx_5 \rightarrow tx_3 \rightarrow tx_4 \rightarrow tx_1 \rightarrow tx_2
\]
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 (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";
}
}
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) }
    }
}
<table>
<thead>
<tr>
<th>The Givens of Smart Contracts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deployed in a low-level language</td>
</tr>
<tr>
<td>Must be <em>Turing-complete</em></td>
</tr>
<tr>
<td>Code is law</td>
</tr>
</tbody>
</table>
The Givens of Smart Contracts

Deployed in a low-level language

Must be *Turing-complete*

Code is law

**Difficult** for audit and verification

Complex semantics, **exploits**

One should understand the **code** to understand the **contract**
Sending a Message or Calling?

```solidity
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;
    }
}
```
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;
    }
}
What’s the Right Model of thinking about Smart Contracts?
Chapter I

The Analogy
A Concurrent Perspective on Smart Contracts

Ilya Sergey
Aquinas Hobor

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

1010    // Burn DAO Tokens
1011    Transfer(msg.sender, 0, balances[msg.sender]);
1012    withdrawRewardFor(msg.sender); // be nice, and get his rewards
1013    totalSupply -= balances[msg.sender];
1014    balances[msg.sender] = 0;
1015    paidOut[msg.sender] = 0;
1016    return true;
1017    }
Reentrancy and multitasking

```
1010  // Burn DAO Tokens
1011  Transfer(msg.sender, 0, balances[msg.sender]);
1012  withdrawRewardFor(msg.sender);  // be nice, and get his rewards
1013  totalSupply -= balances[msg.sender];
1014  balances[msg.sender] = 0;
1015  paidOut[msg.sender] = 0;
1016  return true;
1017  }
```

Fig. 2. DAO code fragment 

Unfortunately, the DAO internal state still indicates that the account is funded since its account balance has not yet been zeroed out in line 1014. Accordingly, a malicious msg.sender can initiate a second withdrawal by calling back into the DAO contract, which will in turn send a second payment when control reaches line 1012 again. In fact, the malicious msg.sender can then initiate a third, fourth, etc. withdrawal, all of which will result in payment. Only at the end is his account zeroed out, after being paid many multiples of its original balance.

Previous analyses of this bug have indicated that the problem is due to recursion or unintended reentrancy. In a narrow sense this is true, but in a wider sense what is going on is that sequential code is running in what is in many senses a concurrent environment.

Interference and Synchronization

Having showed that concurrent-type behavior exists and causes problems in real contracts on the Blockchain, we will now examine other ways that our concurrent-objects-as-contracts viewpoint can help us understand how contracts can behave on the blockchain.

3.1 Atomic updates in shared-memory concurrency

Figure 3 depicts a canonical example (presented in a Java 8-like pseudocode) of a wrongly used concurrent object, which is supposed to implement an "atomic" counter with methods get and set. The implementation of the concurrent counter on the left is obviously thread-safe (i.e., data race-free), thanks to the use of synchronized primitives. What is problematic, though, is how an instance of the Counter class is used in the multithreaded client code on the right. Specifically, with two threads running in parallel and their operations interleaving, the call to incr() within thread2's body could happen, for instance, between the assignment to a and the call c.set(a + 1) within the incr() call of thread1. This would invalidate the condition in the following assert statement, making the overall program fail non-deterministically for a certain execution!

The issue arises because the implementation of incr() on top of Counter does not provide the atomicity guarantees, expected by the client code. Specifically, the code on the right is implemented in the assumption that there will be no interference between the statements of incr(), hence the counter c is going to be incremented by 1, and a and b will be the same by the end of its execution. Indeed,
DAO: withdrawRewardFor()  \(\text{Inv}\) balances[msg.sender] = 0

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

\(\text{Inv}(contract.state, balance)\)

\(\text{Inv}\) \(\text{c.atomicMethod()}\) \(\text{Inv}\) \(\text{Environment}\)  \(\text{Inv}\) \(\text{c.atomicMethod()}\) \(\text{Inv}\) \(\text{Environment}\)
Accounts using *smart contracts* in a blockchain are like threads using *concurrent objects* in shared memory.

<table>
<thead>
<tr>
<th>contract state</th>
<th>—</th>
<th>object state</th>
</tr>
</thead>
<tbody>
<tr>
<td>call/send</td>
<td>—</td>
<td>context switching</td>
</tr>
<tr>
<td>Reentrancy</td>
<td>—</td>
<td>(Un)cooperative multitasking</td>
</tr>
<tr>
<td>Invariants</td>
<td>—</td>
<td>Atomicity</td>
</tr>
</tbody>
</table>
Querying an Oracle

Transaction 1

- c.prepareRequest()
- o.raiseEvent()

Transaction 2

- o.respond()
- c.__callback(data)
Querying an Oracle

Block N

Transaction 1

```
c.prepareRequest()
o.raiseEvent()
```

Block N+M

Transaction 2

```
o.respond()
c.__callback(data)
```
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.

---

**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 MICHAELSKY, Stanford University  
NOAM RINETZKY, Tel Aviv University  
MOLLY SAGIV, Tel Aviv University and VMware Research  
YONI ZOHAR, Tel Aviv University

---

**Exploiting the Laws of Order in Blockchain**

Aashish Kolluri  
School of Computing, NUS  
Singapore  
Ivica Nikolic  
School of Computing, NUS  
Singapore

---

**Finding The Greedy, Prodigal, and Suicidal Contracts at Scale**

Aashish Kolluri  
School of Computing, NUS  
Singapore  
Ilya Sergey  
University College London  
United Kingdom

---

**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)
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 (transition S);
  (* All transitions have unique tags *)
  _ : uniq (map (@tag _) transitions)
  }.

Definition tags {S : Type} (p : Protocol S) :=
  map (@tag _) (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 (s : S)
  pre : cstate S;
  post : cstate S;
  out : option message
}.

Definition trace := seq step

Record Protocol \( (S : \text{Type}) \) :=

\[
\text{CProt} \{ \\
\quad (*\text{Account id} \*) \\
\quad \text{acc : address;} \\
\quad (*\text{Initial balance} \*) \\
\quad \text{init_bal : } \mathbb{N}; \\
\quad (*\text{Initial state of a protocol} \*) \\
\quad \text{init_state : } S; \\
\quad (*\text{Protocol comes with a set of transitions} \*) \\
\quad \text{transitions : seq (transition } S); \\
\quad (*\text{All transitions have unique tags} \*) \\
\quad \_ : \text{uniq (map (@ttag \_\_ ) transitions)} \\
\}\.
\]

Definition tags \( \{S : \text{Type}\} (p : \text{Protocol } S) := \)

\[
\text{map (@ttag \_\_ ) (transitions p).}
\]

End Protocol.

Section Semantics.

Variables \( \{S : \text{Type}\} (p : \text{Protocol } S). \)

(* Blockchain schedules *)

Definition schedule := seq (bstate * message).

(* In a well-formed schedule block numbers only grow *)

Fixpoint wf_sched (sch : schedule) :=

\[
\text{if sch is s :: sch'} \\
\quad \text{then let bnum := block_num s.1 in} \\
\quad \text{all [pred s' | bnum s block_num s'.1] sch' \&\& wf_sched sch'} \\
\quad \text{else true.}
\]

Record step :=

\[
\text{Step \{} \\
\quad \text{pre : cstate } S; \\
\quad \text{post : cstate } S; \\
\quad \text{out : option message} \\
\}\.
\]

Definition trace := seq step
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 (transition S);
    (* All transitions have unique tags *)
    _# : uniq (map (@tag _) transitions)
  }.

Definition tags {S : Type} (p : Protocol S) :=
  map (@tag _) (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
}.
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 (transition S);
    (* All transitions have unique tags *)
    _ : uniq (map (ettag _) transitions)
  }.

Definition tags {S : Type} (p : Protocol S) :=
  map (ettag _) (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' l bnum s 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
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

Account X
Contract Execution Model

- Account X
- Contract D
- Contract C
- Account Z
- Account Y
- Contract E

Edge labels:
- $m_1$
- $m_2$
- $m_3$
- $m_4$
- $m_5$
- $m_6$
Contract Execution Model

Conf\textsubscript{C} \xrightarrow{m_1} Conf'\textsubscript{C} \xrightarrow{m_6} Conf''\textsubscript{C}

Conf\textsubscript{D} \xrightarrow{m_2} Conf'\textsubscript{D}

Conf\textsubscript{E} \xrightarrow{m_4} Conf'\textsubscript{E}

Final contract states

Fixed MAX length of call sequence

Final contract states

Fixed MAX length of call sequence
Contract Execution Model

Conf_C \xrightarrow{m_1} \text{Conf}_C' \xrightarrow{m_6} \text{Conf}_C''
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

\[ Q \text{ since } P \text{ as long } R \equiv \forall \text{ conf conf}', \text{ conf } \rightarrow_{R^*} \text{ conf}', P(\text{conf}) \Rightarrow Q(\text{conf, conf}') \]

- “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

\[ Q \text{ since } P \text{ as long } R \quad \overset{\text{def}}{=} \quad \forall \text{ conf conf}', \text{ conf } \rightarrow_{R^*} \text{ conf}', \text{ } P(\text{conf}) \Rightarrow Q(\text{conf}, \text{conf}') \]

**Definition** since\_as\_long

\[
(P : \text{conf} \rightarrow \text{Prop}) \\
(Q : \text{conf} \rightarrow \text{conf} \rightarrow \text{Prop}) \\
(R : \text{bstate} \times \text{message} \rightarrow \text{Prop}) := \\
\forall \text{ sc conf conf}', \text{ } P \text{ st } \\
\text{ (conf } \leadsto \text{ conf' sc) } \land (\forall \text{ b, } \text{b } \in \text{ sc } \rightarrow R \text{ b) } \rightarrow \\
Q \text{ conf conf'}. 
\]
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 \((b : \text{address})\) \((d : \text{amount})\) \(\text{conf} :=\)
\[
\text{conf.backers}(b) == d.
\]

**Definition** no_claims_from \((b : \text{address})\)
\((q : \text{bstate * message}) :=\)
\[
q.message.sender != b.
\]

**Lemma** donation_preserved \((b : \text{address})\) \((d : \text{amount})\):
\[
\text{since.as long (donated b d) (fun c c' => donated b d c') (no_claims_from b).}
\]

\(b\)’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

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

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

Chaodong Zheng  
National University of Singapore  
chaodong.zheng@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

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

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

Chaodong Zheng
National University of Singapore
chaodong.zheng@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 enquery@zilliqa.com @zilliqa
Sure, I’d love to!

We’re building this cool sharded blockchain. Would you like to help us a language for provably safe smart contracts?

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
<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Computations</td>
<td>self-explanatory</td>
</tr>
<tr>
<td>State Manipulation</td>
<td>changing contract's fields</td>
</tr>
<tr>
<td>Effects</td>
<td>accepting funds, logging events</td>
</tr>
<tr>
<td>Communication</td>
<td>sending funds, calling other contracts</td>
</tr>
</tbody>
</table>
Verified Specification

Communication

Verified Specification

State Manipulation
Effects

Verified Specification

Computations
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

(signed integers) \( int ::= \ i32 \mid i64 \mid i128 \mid i256 \)

(unsigned integers) \( uint ::= \ u32 \mid u64 \mid u128 \mid u256 \)

(byte strings) \( bst ::= \ bystrx \ n \mid bystr \)

(primitive types) \( pt ::= \ int \mid uint \mid bst \mid\)
\( \quad \text{string} \mid bnum \mid msg \)

(algebraic types) \( D ::= \ unit \mid bool \mid nat \mid option \mid\)
\( \quad \text{pair} \mid list \mid U \)

(general Types) \( t ::= \ pt \mid map \ t \ t \mid t \to t \mid\)
\( \quad D \ t \mid \alpha \mid \forall \alpha. \ t \)
### Expressions (pure)

<table>
<thead>
<tr>
<th>Expression</th>
<th>$e$ ::=</th>
<th>$f$</th>
<th>simple expression</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$\text{let } x \langle : T \rangle = f \text{ in } e$</td>
<td>let-form</td>
</tr>
<tr>
<td>Simple expression</td>
<td>$f ::= l$</td>
<td>primitive literal</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$x$</td>
<td>variable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>${ \langle \text{entry} \rangle_k }$</td>
<td>Message</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{fun } (x : T) \Rightarrow e$</td>
<td>function</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{builtin } b \langle x_k \rangle$</td>
<td>built-in application</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$x \langle x_k \rangle$</td>
<td>application</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{tfun } \alpha \Rightarrow e$</td>
<td>type function</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{@x } T$</td>
<td>type instantiation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$C \langle \langle T_k \rangle \rangle \langle x_k \rangle$</td>
<td>constructor instantiation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{match } x \text{ with } \langle \mid \text{sel}_k \rangle \text{ end}$</td>
<td>pattern matching</td>
<td></td>
</tr>
<tr>
<td>Selector</td>
<td>$sel ::= \text{pat} \Rightarrow e$</td>
<td>variable binding</td>
<td></td>
</tr>
<tr>
<td>Pattern</td>
<td>$pat ::= x$</td>
<td>constructor pattern</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$C \langle \text{pat}_k \rangle$</td>
<td>parenthesized pattern</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( pat )</td>
<td>wildcard pattern</td>
<td></td>
</tr>
<tr>
<td>Message entry</td>
<td>$entry ::= b : x$</td>
<td>identifier</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>$b$</td>
<td>identifier</td>
<td></td>
</tr>
</tbody>
</table>
Structural Recursion in Scilla

Natural numbers (not Ints!)

$$\text{nat}_\text{rec} : \forall \alpha. \alpha \rightarrow (\text{nat} \rightarrow \alpha \rightarrow \alpha) \rightarrow \text{nat} \rightarrow \alpha$$

- Result type
- Value for 0
- Constructing the next value
- Number of iterations
- Final result
Example: Fibonacci Numbers

```haskell
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

```ocaml
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
  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
```
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

1. let fib = fun (n : Nat) =>
2. let iter_nat = @ nat_rec (Pair Int Int) in
3. let iter_fun =
4. 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
5. let zero = 0 in
6. let one = 1 in
7. let init_val = And {Int Int} one zero in
8. let res = iter_nat init_val iter_fun n in
9. 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
  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

```plaintext
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 y
      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
list_foldr: $\forall 'A 'B. ('A \to 'B \to 'B) \to 'B \to (List 'A) \to 'B$

- **Element type**
- **Result type**
- **Iterator for non-empty list**
- **Value for Nil**
- **Argument list**
- **Result**
More Structural Recursion with Lists

list_foldk: \( \forall \ 'A \ 'B. (\ 'B \to \ 'A \to (\ 'B \to \ 'B) \to \ 'B) \to \)

\( \ 'B \to (\text{List} \ 'A) \to \ 'B \)
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)

\[
s ::= \ x <- f \\
  f ::= x \\
  x = e \\
  \text{match } x \text{ with } \langle \text{pat } => s \rangle \text{ end} \\
  x <- \ &B \\
  \text{accept} \\
  \text{event } m \\
  \text{send } ms \\
  \text{throw} \\
  \text{in-place map operations}
\]

read from mutable field
store to a field
assign a pure expression
pattern matching and branching
read from blockchain state
accept incoming payment
create a single event
send list of messages
abort the execution
efficient manipulation with maps
Statement Semantics

$[s] : \text{BlockchainState} \rightarrow \text{Configuration} \rightarrow \text{Configuration}$

$\text{BlockchainState}$: Immutable global data (block number etc.)

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

- Immutable bindings
- Contract's own funds
- Messages to be sent
- Mutable fields
- Funds sent to contract
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 \((sender: Address, amount: Int)\)

\[
\text{blk} <- \text{& BLOCKNUMBER};
\]
\[
in\_time = \text{blk}\_leq \text{blk max\_block};
\]
match \(in\_time\) with

\[
| \text{True} => \\
\text{bs} <- \text{backers};
\res = \text{check\_update bs sender amount};
\text{match res with}
\]

\[
| \text{None} => \\
\text{msg} = \{\text{tag : Main; to : sender; amount : 0; code : already\_backed};
\text{msgs} = \text{one\_msg msg};
\text{send msgs}
\]

\[
| \text{Some} bs1 => \\
\text{backers} := bs1;
\text{accept};
\text{msg} = \{\text{tag : Main; to : sender; amount : 0; code : accepted\_code};
\text{msgs} = \text{one\_msg msg};
\text{send msgs}
\end
\]

\[
| \text{False} => \\
\text{msg} = \{\text{tag : Main; to : sender; amount : 0; code : missed\_dealine};
\text{msgs} = \text{one\_msg msg};
\text{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
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
    | 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)
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_dealline}; |
| 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
                | False =>
                    msg = {tag : Main; to : sender; amount : 0; code : missed_dealine};
                    msgs = one_msg msg;
                    send msgs
            end
        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

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

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
Contract Structure

Library of pure functions

Immutable parameters

Mutable fields

Transition 1

...

Transition N
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: \((\text{size of patterns}) \times (\text{number of branches})\)
- 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

- 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

<table>
<thead>
<tr>
<th>Contract</th>
<th>LOC</th>
<th>#Lib</th>
<th>#Trans</th>
</tr>
</thead>
<tbody>
<tr>
<td>HelloWorld</td>
<td>31</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Crowdfunding</td>
<td>127</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>Auction</td>
<td>140</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>ERC20</td>
<td>158</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>ERC721</td>
<td>270</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>Wallet</td>
<td>363</td>
<td>28</td>
<td>9</td>
</tr>
<tr>
<td>Bookstore</td>
<td>123</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>HashGame</td>
<td>209</td>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>Schnorr</td>
<td>71</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
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
Microsoft Research
dimtris, simonpj}@microsoft.com

Simon Peyton Jones
Gas Usage Analysis

\[
(* \text{forall 'A. forall 'B. ('A} \rightarrow \text{'B)} \rightarrow \text{List 'A} \rightarrow \text{List 'B} *)
\]

\[
\text{let list_map} = \text{tfun 'A} \Rightarrow \text{tfun 'B} \Rightarrow
\]

\[
\text{fun (f : 'A} \rightarrow \text{'B)} \Rightarrow \text{fun (l : List 'A)} \Rightarrow
\]

\[
\text{let folder = @list_foldr 'A (List 'B) in}
\]

\[
\text{let init = Nil 'B} \text{ in}
\]

\[
\text{let iter = fun (h : 'A)} \Rightarrow \text{fun (z : List 'B)} \Rightarrow
\]

\[
\text{let h1 = f h in}
\]

\[
\text{Cons 'B} \text{ h1 z}
\]

\[
\text{in folder iter init l}
\]

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**

<table>
<thead>
<tr>
<th></th>
<th>Money</th>
<th>NotMoney</th>
<th>Map τ</th>
<th>t t</th>
<th>T</th>
<th>⊥</th>
</tr>
</thead>
<tbody>
<tr>
<td>t ::=</td>
<td>Option</td>
<td>Pair</td>
<td>List</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- (maps) $\text{Map } τ \sqsubseteq \text{Map } τ'$ iff $τ \sqsubseteq τ'$
- (algebraic types) $t t \sqsubseteq t' t'$ iff $t = t'$ and $τ_i \sqsubseteq τ'_i$ for all $i$
- (bottom) $⊥ \sqsubseteq τ$ for all $τ$
- (top) $τ \sqsubseteq T$ for all $τ$

**Results for Crowdfunding**

<table>
<thead>
<tr>
<th>Field/Param</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>NotMoney</td>
</tr>
<tr>
<td>max_block</td>
<td>NotMoney</td>
</tr>
<tr>
<td>goal</td>
<td>Money</td>
</tr>
<tr>
<td>backers</td>
<td>Map Money</td>
</tr>
<tr>
<td>funded</td>
<td>NotMoney</td>
</tr>
</tbody>
</table>
### Analysis Results

<table>
<thead>
<tr>
<th>Contract</th>
<th>LOC</th>
<th>#Lib</th>
<th>#Trans</th>
</tr>
</thead>
<tbody>
<tr>
<td>HelloWorld</td>
<td>31</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Crowdfunding</td>
<td>127</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>Auction</td>
<td>140</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>ERC20</td>
<td>158</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>ERC721</td>
<td>270</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>Wallet</td>
<td>363</td>
<td>28</td>
<td>9</td>
</tr>
<tr>
<td>Bookstore</td>
<td>123</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>Bookstore</td>
<td>123</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>HashGame</td>
<td>209</td>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>Schnorr</td>
<td>71</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
### Gas Usage Analysis

<table>
<thead>
<tr>
<th>Contract</th>
<th>LOC</th>
<th>#Lib</th>
<th>#Trans</th>
<th>Asympt. GU</th>
</tr>
</thead>
<tbody>
<tr>
<td>HelloWorld</td>
<td>31</td>
<td>3</td>
<td>2</td>
<td>$O(</td>
</tr>
<tr>
<td>Crowdfunding</td>
<td>127</td>
<td>13</td>
<td>3</td>
<td>$O(</td>
</tr>
<tr>
<td>Auction</td>
<td>140</td>
<td>11</td>
<td>3</td>
<td>$O(</td>
</tr>
<tr>
<td>ERC20</td>
<td>158</td>
<td>2</td>
<td>6</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>ERC721</td>
<td>270</td>
<td>15</td>
<td>6</td>
<td>$O(</td>
</tr>
<tr>
<td>Wallet</td>
<td>363</td>
<td>28</td>
<td>9</td>
<td>$O(</td>
</tr>
<tr>
<td>Bookstore</td>
<td>123</td>
<td>6</td>
<td>3</td>
<td>$O(</td>
</tr>
<tr>
<td>HashGame</td>
<td>209</td>
<td>16</td>
<td>3</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
| Schnorr         | 71  | 2    | 3      | $O(|\text{bystr}|)$
Cash-Flow Analysis

<table>
<thead>
<tr>
<th>Contract</th>
<th>LOC</th>
<th>#Lib</th>
<th>#Trans</th>
<th>Asympt. GU</th>
<th>$-Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>HelloWorld</td>
<td>31</td>
<td>3</td>
<td>2</td>
<td>(O(</td>
<td>\text{string}</td>
</tr>
<tr>
<td>Crowdfunding</td>
<td>127</td>
<td>13</td>
<td>3</td>
<td>(O(</td>
<td>\text{map}</td>
</tr>
<tr>
<td>Auction</td>
<td>140</td>
<td>11</td>
<td>3</td>
<td>(O(</td>
<td>\text{map}</td>
</tr>
<tr>
<td>ERC20</td>
<td>158</td>
<td>2</td>
<td>6</td>
<td>(O(1))</td>
<td>✓*</td>
</tr>
<tr>
<td>ERC721</td>
<td>270</td>
<td>15</td>
<td>6</td>
<td>(O(</td>
<td>\text{map}</td>
</tr>
<tr>
<td>Wallet</td>
<td>363</td>
<td>28</td>
<td>9</td>
<td>(O(</td>
<td>\text{map}</td>
</tr>
<tr>
<td>Bookstore</td>
<td>123</td>
<td>6</td>
<td>3</td>
<td>(O(</td>
<td>\text{string}</td>
</tr>
<tr>
<td>HashGame</td>
<td>209</td>
<td>16</td>
<td>3</td>
<td>(O(1))</td>
<td>✓</td>
</tr>
<tr>
<td>Schnorr</td>
<td>71</td>
<td>2</td>
<td>3</td>
<td>(O(</td>
<td>\text{bystr}</td>
</tr>
</tbody>
</table>
## Cash-Flow Analysis

<table>
<thead>
<tr>
<th>Contract</th>
<th>LOC</th>
<th>#Lib</th>
<th>#Trans</th>
<th>Asympt. GU</th>
<th>$-Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>HelloWorld</td>
<td>31</td>
<td>3</td>
<td>2</td>
<td>$O(</td>
<td>\text{string}</td>
</tr>
<tr>
<td>Crowdfunding</td>
<td>127</td>
<td>13</td>
<td>3</td>
<td>$O(</td>
<td>\text{map}</td>
</tr>
<tr>
<td>Auction</td>
<td>140</td>
<td>11</td>
<td>3</td>
<td>$O(</td>
<td>\text{map}</td>
</tr>
<tr>
<td><strong>ERC20</strong></td>
<td>158</td>
<td>2</td>
<td>6</td>
<td>$O(1)$</td>
<td>✓*</td>
</tr>
<tr>
<td>ERC721</td>
<td>270</td>
<td>15</td>
<td>6</td>
<td>$O(</td>
<td>\text{map}</td>
</tr>
<tr>
<td>Wallet</td>
<td>363</td>
<td>28</td>
<td>9</td>
<td>$O(</td>
<td>\text{map}</td>
</tr>
<tr>
<td>Bookstore</td>
<td>123</td>
<td>6</td>
<td>3</td>
<td>$O(</td>
<td>\text{string}</td>
</tr>
<tr>
<td>HashGame</td>
<td>209</td>
<td>16</td>
<td>3</td>
<td>$O(1)$</td>
<td>✓</td>
</tr>
<tr>
<td>Schnorr</td>
<td>71</td>
<td>2</td>
<td>3</td>
<td>$O(</td>
<td>\text{bystr}</td>
</tr>
</tbody>
</table>

* ✓ means the contract is safe for $\mathcal{S}$ with $\mathcal{F}$ implemented, ✓* means additional analysis is needed, ✓| means the contract is safe for $\mathcal{S}$ with $\mathcal{F}$ implemented and overall $\mathcal{S}$. 

### Table Notes:
- **LOC**: Lines of Code
- **#Lib**: Number of Libraries
- **#Trans**: Number of Transitions
- **Asympt. GU**: Asymptotic Guarenteed
- **$-Flow$**: Safe Flow

### Tokens:
- **Fungible Tokens**: Similar to traditional currencies, where coins are indistinguishable and freely interchangeable.
- **Non-fungible Tokens**: Similar to collector items, where tokens are distinguishable and may not be interchangeable.

### Additional Observations:
- Approximately 16% of active smart contracts on Ethereum are ERC20.
- According to https://etherscan.io/tokens, roughly 16% of active smart contracts on Ethereum are ERC20.
- ERC20's performance on common contracts and what are the main bottlenecks?
- Table reports on ERC20, ERC721, Wallet, Bookstore, HashGame, Schnorr, and their corresponding LOC, #Lib, #Trans, Asympt. GU, and $-Flow$.
# Cash-Flow Analysis

<table>
<thead>
<tr>
<th>Contract</th>
<th>LOC</th>
<th>#Lib</th>
<th>#Trans</th>
<th>Asympt. GU</th>
<th>$-Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>HelloWorld</td>
<td>31</td>
<td>3</td>
<td>2</td>
<td>O(</td>
<td>string</td>
</tr>
<tr>
<td>Crowdfunding</td>
<td>127</td>
<td>13</td>
<td>3</td>
<td>O(</td>
<td>map</td>
</tr>
<tr>
<td>Auction</td>
<td>140</td>
<td>11</td>
<td>3</td>
<td>O(</td>
<td>map</td>
</tr>
<tr>
<td>ERC20</td>
<td>158</td>
<td>2</td>
<td>6</td>
<td>O(1)</td>
<td>✓*</td>
</tr>
<tr>
<td>ERC721</td>
<td>270</td>
<td>15</td>
<td>6</td>
<td>O(</td>
<td>map</td>
</tr>
<tr>
<td>Wallet</td>
<td>363</td>
<td>28</td>
<td>9</td>
<td>O(</td>
<td>map</td>
</tr>
<tr>
<td>Bookstore</td>
<td>123</td>
<td>6</td>
<td>3</td>
<td>O(</td>
<td>string</td>
</tr>
<tr>
<td>HashGame</td>
<td>209</td>
<td>16</td>
<td>3</td>
<td>O(1)</td>
<td>✓</td>
</tr>
<tr>
<td>Schnorr</td>
<td>71</td>
<td>2</td>
<td>3</td>
<td>O(</td>
<td>bystr</td>
</tr>
</tbody>
</table>

The table reports on the LOC, number of libraries (#Lib), number of transitions (#Trans), asymptotic gas usage (Asympt. GU), and the $-Flow. Note that ERC721 contracts are highlighted in red indicating non-fungible tokens.
For our evaluation we have chosen the most common kinds of contracts used on Ethereum: ERC20 (ft), ERC721 (nft), auction (auc) and crowdfunding (cfd). Performance experiments were conducted on a commodity Intel Core i5 machine with 8GB RAM.

To answer question (1), we have evaluated the interpreter performance on the most expensive transitions of the chosen contracts (e.g., ERC20’s transfer), with the size of the largest affected contract state component (e.g., a map field) ranging from 10k to 500k entries. The results are shown in Tab. 3 and Fig. 12a. It is clear that the evaluator’s performance overhead is negligible (less than 1%) compared to the time taken by input/output of the contract state: reading from blockchain (init), serialising and writing it back—those machineries operate with JSON representation of state and their performance deteriorates linearly with the state size. This issue is orthogonal to our study of the language design presented in this paper, and in Sec. 7, we discuss possible ways to address it in the future. That said, even with the suboptimal IO implementation, in most of the cases the observed transaction times are under 10s, which is acceptable for blockchain computations.

The implementation of Scilla is agnostic with regard to the underlying blockchain protocol, and at the moment all interaction is done by passing state snapshots in JSON. Thus, making an apples-to-apples comparison of Scilla/EVM performance is difficult, as EVM is an integral part of the Ethereum protocol, and can access the entire blockchain state in a RAM-like manner. This leads to more slow start-up time for EVM, but nearly constant-time access for contracts with large state, whereas Scilla input-output overhead grows linearly. Fig. 12b shows a comparison of run-times (from the cold start) of Scilla and EVM on the same four contracts with 10k and 50k state entries (first/second four groups). In most of the cases, Scilla’s performance is better, but EVM shows superior results, due to more efficient IO, when the state grows beyond 50k entries. The state of nft is larger than the projected 10/50k, as it uses nested maps, while we only count “top-level” entries.

The artefact containing the benchmarks is available on GitHub: https://github.com/ilyasergey/scilla-benchmarks.

The largest Ethereum contract to date is ERC20 with 600k entries. Most of deployed contracts have less than 50k entries.
Performance

Table 3. Breakdown of contract run-times (in ms): initialisation, execution, serialisation, and output.

<table>
<thead>
<tr>
<th>Transition/State size</th>
<th>ft-transfer</th>
<th>nft-setApproveForAll</th>
<th>auc-bid</th>
<th>cfd-pledge</th>
</tr>
</thead>
<tbody>
<tr>
<td>10k</td>
<td>67</td>
<td>239</td>
<td>61</td>
<td>68</td>
</tr>
<tr>
<td>100k</td>
<td>709</td>
<td>3,011</td>
<td>665</td>
<td>723</td>
</tr>
<tr>
<td>500k</td>
<td>4,208</td>
<td>15,382</td>
<td>3,480</td>
<td>3,705</td>
</tr>
</tbody>
</table>

Time (s)

(a) Relative time breakdown

(b) Scilla/EVM execution times

(c) Code size comparison

For our evaluation we have chosen the most common kinds of contracts used on Ethereum: ERC20 (ft), ERC721 (nft), auction (auc), and crowdfunding (cfd). Performance experiments were conducted on a commodity Intel Core i5 machine with 8GB RAM.

To answer question (1), we have evaluated the interpreter performance on the most expensive transitions of the chosen contracts (e.g., ERC20's transfer), with the size of the largest affected contract state component (e.g., a map field) ranging from 10k to 500k entries. The results are shown in Tab. 3 and Fig. 12a. It is clear that the evaluator's performance overhead is negligible (less than 1%) compared to the time taken by input/output of the contract state: reading from blockchain (init), serialising and writing it back—those machineries operate with JSON representation of state and their performance deteriorates linearly with the state size. This issue is orthogonal to our study of the language design presented in this paper, and in Sec. 7, we discuss possible ways to address it in the future. That said, even with the suboptimal IO implementation, in most of the cases the observed transaction times are under 10s, which is acceptable for blockchain computations.

The implementation of Scilla is agnostic with regard to the underlying blockchain protocol, and at the moment all interaction is done by passing state snapshots in JSON. Thus, making an apples-to-apples comparison of Scilla/EVM performance is difficult, as EVM is an integral part of the Ethereum protocol, and can access the entire blockchain state in a RAM-like manner. This leads to more slow start-up time for EVM, but nearly constant-time access for contracts with large state, whereas Scilla input-output overhead grows linearly. Fig. 12b shows a comparison of run-times (from the cold start) of Scilla and EVM on the same four contracts with 10k and 50k state entries (first/second four groups). In most of the cases, Scilla's performance is better, but EVM shows superior results, due to more efficient IO, when the state grows beyond 50k entries. The state of nft is larger than the projected 10/50k, as it uses nested maps, while we only count "top-level" entries.

Chapter 5

The Challenges
Maps that Grow

Who has donated

<table>
<thead>
<tr>
<th>Donor</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amrit</td>
<td>100</td>
</tr>
<tr>
<td>Jacob</td>
<td>120</td>
</tr>
<tr>
<td>Vaivas</td>
<td>500000</td>
</tr>
</tbody>
</table>

The table can grow very large!
Initial Naïve Execution Model

<table>
<thead>
<tr>
<th>Donor</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amrit</td>
<td>100</td>
</tr>
<tr>
<td>Jacob</td>
<td>120</td>
</tr>
<tr>
<td>Vaivas</td>
<td>500000</td>
</tr>
</tbody>
</table>

Anton donates

Core Protocol

Scilla

DB
Initial Naïve Execution Model

<table>
<thead>
<tr>
<th>Donor</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amrit</td>
<td>100</td>
</tr>
<tr>
<td>Jacob</td>
<td>120</td>
</tr>
<tr>
<td>Vaivas</td>
<td>500000</td>
</tr>
<tr>
<td>Anton</td>
<td>10000</td>
</tr>
</tbody>
</table>

Anton donates

Core Protocol

Scilla

DB
Fine-Grained Interaction

Core ----> Scilla

Amrit?
100
Anton = 10000

DB
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
import BoolUtils

library Crowdfunding

let one_msg = fun (msg : Message) =>
  let nil_msg = Nil {Message in}
  Cons {Message in} 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 bcl = builtin put bs _sender _amount in
                Some (Map ByStr20 Uint128) bcl
      | True => None (Map ByStr20 Uint128)
      end

let blk_len = fun (blk1 : BNun) =>
  fun (blk2 : BNun) =>
    let bcl = builtin blk blk1 blk2 in
    let bcl = builtin eq blk1 blk2 in
    orb bcl bcl

let accepted_code = Int32 1
let missed_deadline_code = Int32 2
let already_backed_code = Int32 3
let not_owner_code = Int32 4
Contract

zil1w0g7tnxk8usu6et44jfcuwh0mjc040pqd6l

**Balance**: 193.35 ZIL

**Transactions**: 40

**Contract Creation**

zil1fxx... at dab4e4878c0d93abcfaa...

---

```
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 code: library *)
let code_success = Uint32 0
```
Scilla on a Sharded Blockchain
TYPES OF TRANSACTIONS

TYPE I
$5

TYPE II
foo(x)

TYPE III
foo(x) bar(x) $5
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
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?
Thanks!

http://scilla-lang.org

OOPSLA’19

Safer Smart Contract Programming with ScILLA

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

TTECH, Indore, India

K.N. Reals, Aarhus University, Denmark

British University, United Kingdom

Alvin A. Ivanov, Institute of Information Technology, Russia

Abani S. Choudhury, University of Malaya, Research, Malaysia
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.