The rise of programmable open distributed consensus platforms based on the blockchain technology has aroused a lot of interest in replicated stateful computations, aka smart contracts. As blockchains are used predominantly in financial applications, smart contracts frequently manage millions of dollars worth of virtual coins. Since smart contracts cannot be updated once deployed, the ability to reason about their correctness becomes a critical task. Yet, the de facto implementation standard, pioneered by the Ethereum platform, dictates smart contracts to be deployed in a low-level language, which renders independent audit and formal verification of deployed code infeasible in practice.

We report an ongoing experiment held with an industrial blockchain vendor on designing, evaluating, and deploying Scilla, a new programming language for safe smart contracts. Scilla is positioned as an intermediate-level language, suitable to serve as a compilation target and also as an independent programming framework. Taking System F as a foundational calculus, Scilla offers strong safety guarantees by means of type soundness. It provides a clean separation between pure computational, state-manipulating, and communication aspects of smart contracts, avoiding many known pitfalls due to execution in a byzantine environment. We describe the motivation, design principles, and semantics of Scilla, and we report on Scilla use cases provided by the developer community. Finally, we present a framework for lightweight verification of Scilla programs, and showcase it with two domain-specific analyses on a suite of real-world use cases.

CCS Concepts:
• Software and its engineering → Functional languages; Distributed programming
• Theory of computation → Program analysis.

Additional Key Words and Phrases: Blockchain, Smart Contracts, Domain-Specific Languages, Static Analysis

1 INTRODUCTION

Smart contracts are self-enforcing, self-executing protocols governing an interaction between several (mutually distrusting) parties. Initially proposed by Szabo (1994), this idea could only be implemented in a practical setting more than fifteen years later, with the rise of open byzantine consensus protocols powered by the blockchain technology (Bano et al. 2017; Pîrlea and Sergey).
Blockchains 101

- 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

\[ tx_5 \rightarrow tx_3 \rightarrow tx_4 \rightarrow tx_1 \rightarrow tx_2 \]
Blockchains 101

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

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

\[ tx_5 \rightarrow tx_3 \rightarrow tx_4 \rightarrow tx_1 \rightarrow tx_2 \]
Blockchains 101

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

\[\begin{align*}
[tx_5, tx_3] & \leftarrow [tx_4] \leftarrow [tx_1, tx_2] \\
& \downarrow \\
& tx_5 \rightarrow tx_3 \rightarrow tx_4 \rightarrow tx_1 \rightarrow tx_2
\end{align*}\]
Transactions

- Executed *locally*, alter the *replicated* state.

- Simplest case: *transferring funds* from A to B, **consensus**: *no* double spending.

- More interesting: deploying and executing *replicated computations*.
Smart Contracts

• *Stateful mutable* objects replicated via a consensus protocol
• State typically involves a stored amount of *funds/currency*
• Main usages:
  • crowdfunding and ICO
  • multi-party accounting
  • voting and arbitration
  • puzzle-solving games with distribution of rewards
• Supporting platforms: *Ethereum, Tezos, Concordium, FB Libra,*…
A Smart Contract in Solidity™

```solidity
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";
    }
}
```
## The Givens of Smart Contracts

<table>
<thead>
<tr>
<th>Deployed in a <em>low-level language</em></th>
<th>Uniform compilation target</th>
</tr>
</thead>
<tbody>
<tr>
<td>Must be <em>Turing-complete</em></td>
<td>Run arbitrary computations</td>
</tr>
<tr>
<td>Code is law</td>
<td>What else if not the code?</td>
</tr>
</tbody>
</table>
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?

```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;
    }
}
```
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;
    }
}
```

Caller can reenter and withdraw again
Tomorrow

11:00 - 12:30: OOPSLA - Repair & Transformation at Templars
Chair(s): Bor-Yuh Evan Chang
University of Colorado Boulder | Amazon

11:00 - 11:22  Detected Nondeterministic Payment Bugs in Ethereum Smart Contracts
Shuai Wang ETH Zurich, Chengyu Zhang East China Normal University, Zhendong Su ETH Zurich

11:22 - 11:45  Automatic Repair of Regular Expressions
Rong Pan University of Texas at Austin, Qinheping Hu University of Wisconsin, Madison, Gaowei Xu University of Wisconsin Madison

Talk

DOI
Pre-print
The Challenge

Preventing smart contract vulnerabilities with principled Programming Language design
Wishlist

• Explicit interaction: no reentrancy attacks
• Minimalistic
• Explicit control of effects
• Expressive
• Analysis/Verification friendly
• Predictable resource (gas) consumption
• Reasonable performance
The Givens of Smart Contracts

Deployed in a low-level language

Must be Turing-complete

Code is law (so it should be easy to interpret)
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

Simple computation model
Not Turing-complete
Explicit Effects
Communication

System F with small extensions
Only primitive recursion/iteration
State-transformer semantics
Contracts are Autonomous Actors
Smart Contracts as Autonomous Actors
Scilla Contract Execution Model

Account X
Scilla Contract Execution Model

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

Flow paths:
- m₁ from Account X to Contract C
- m₂ from Contract D to Contract C
- m₃ from Contract C to Account Y
- m₄ from Contract C to Contract E
- m₅ from Account Z to Contract D
- m₆ from Contract E to Contract C
Scilla Contract Execution Model

\[ \text{Conf}_C \xrightarrow{m_1} \text{Conf}'_C \xrightarrow{m_6} \text{Conf}''_C \]
\[ \text{Conf}_D \xrightarrow{m_2} \text{Conf}'_D \]
\[ \text{Conf}_E \xrightarrow{m_4} \text{Conf}'_E \]
Scilla Contract Execution Model

$$\text{Conf}_C \xrightarrow{m_1} \text{Conf}'_C \xrightarrow{m_6} \text{Conf}''_C$$
Wishlist

- **Explicit interaction**: no reentrancy attacks
- Minimalistic
- Explicit control of effects
- Expressive
- Analysis/Verification friendly
- Predictable resource (gas) consumption
- Reasonable performance
Wishlist

• Explicit interaction: no reentrancy attacks

• Minimalistic

• Explicit control of effects

• Expressive

• Analysis/Verification friendly

• Predictable resource (gas) consumption

• Reasonable performance
Wishlist

- Explicit interaction: no reentrancy attacks
- Minimalistic
  - Explicit control of effects
- Expressive
- Analysis/Verification friendly
- Predictable resource (gas) consumption
- Reasonable performance
Types

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

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

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

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

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

(general Types) \( \text{t} ::= \text{pt} \mid \text{map t t} \mid \text{t \rightarrow t} \mid \)
\( \mathcal{D} \bar{t} \mid \alpha \mid \text{forall } \alpha . \ t \)
Expressions (pure)

<table>
<thead>
<tr>
<th>Expression</th>
<th>$e$</th>
<th>$::=$</th>
<th>$f$</th>
<th>simple expression</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>$let$ $x$ $::=$ $T$ $=$ $f$ $in$ $e$</td>
<td>let-form</td>
</tr>
<tr>
<td>Simple expression</td>
<td>$f$</td>
<td>$::=$</td>
<td>$l$</td>
<td>primitive literal</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$x$</td>
<td>variable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>${\langle entry\rangle_k}$</td>
<td>Message</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$fun$ $\langle x : T \rangle$ $=&gt;$ $e$</td>
<td>function</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$builtin$ $b$ $\langle x_k \rangle$</td>
<td>built-in application</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$x$ $\langle x_k \rangle$</td>
<td>application</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$tfun$ $\alpha$ $=&gt;$ $e$</td>
<td>type function</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$@x$ $T$</td>
<td>type instantiation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$C$ $\langle {\langle T_k\rangle} \rangle$ $\langle x_k \rangle$</td>
<td>constructor instantiation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$match$ $x$ $with$ $\langle$ $</td>
<td>$ $sel_k$ $\rangle$ $end$</td>
</tr>
<tr>
<td>Selector</td>
<td>$sel$</td>
<td>$::=$</td>
<td>$pat$ $=&gt;$ $e$</td>
<td>variable binding</td>
</tr>
<tr>
<td>Pattern</td>
<td>$pat$</td>
<td>$::=$</td>
<td>$x$</td>
<td>constructor pattern</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$C$ $\langle$ $pat_k$ $\rangle$</td>
<td>paranthesized pattern</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\langle$ $pat$ $\rangle$</td>
<td>wildcardized pattern</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>_</td>
<td>identifier</td>
</tr>
<tr>
<td>Message entry</td>
<td>$entry$</td>
<td>$::=$</td>
<td>$b : x$</td>
<td>identifier</td>
</tr>
<tr>
<td>Name</td>
<td>$b$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Structural Recursion in Scilla

Natural numbers (not Ints!)

\[
\text{nat_rec}: \forall \alpha. \alpha \rightarrow (\text{nat} \rightarrow \alpha \rightarrow \alpha) \rightarrow \text{nat} \rightarrow \alpha
\]

- Result type
- Value for 0
- constructing the next value
- number of iterations
- final result
Example: Fibonacci Numbers

```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
```
Statements (effectful)

s ::=  
  x <- f  
  f := x  
  x = e  
  match x with 〈pat => s〉 end  
  x <- &B  
  accept  
  event m  
  send ms  
  throw  
  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
Statements (effectful)

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

- 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
Statements (effectful)

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

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
Statements (effectful)

s ::= x <- f
f := x
x = e
match x with (pat => s) end
x <- &B
accept event m
send ms
throw
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
Statements (effectful)

s ::= x <- f
f := x
x = e
match x with (pat => s) end
x <- &B
accept event m
send ms
throw
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
Statements (effectful)

\[ s ::= \]
\[
  x \leftarrow f \\
  f := x \\
  x = e \\
  \text{match } x \text{ with } \langle \text{pat} \Rightarrow s \rangle \text{ end} \\
  x \leftarrow \&B \\
  \text{accept} \\
  \text{event } m \\
  \text{send } ms \\
  \text{throw}
\]

\textit{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
Wishlist

- Explicit interaction: no reentrancy attacks
- Minimalistic
  - Explicit control of effects
  - Expressive
- Analysis/Verification friendly
- Predictable resource (gas) consumption
- Reasonable performance
Wishlist

- Explicit interaction: no reentrancy attacks
- Minimalistic (core interpreter ~200 LOC of OCaml)
  - Explicit control of effects
  - Expressive
  - Analysis/Verification friendly
  - Predictable resource (gas) consumption
  - Reasonable performance
Wishlist

- Explicit interaction: no reentrancy attacks
- **Minimalistic** (core interpreter ~200 LOC of OCaml)
  - Explicit control of effects
  - Expressive
  - Analysis/Verification friendly
  - Predictable resource (gas) consumption
  - Reasonable performance
Wishlist

• Explicit interaction: no reentrancy attacks
• Minimalistic (core interpreter ~200 LOC of OCaml)
• Explicit control of effects
  • Expressive
  • Analysis/Verification friendly
• Predictable resource (gas) consumption
• Reasonable performance
Contract Structure

1 library Crowdfunding
2 (* Map ByStr20 Uint128 → ByStr20 → Uint128 → *)
3 (* Option (Map ByStr20 Uint128) *)
4 let check_update = (* ... *)
5 (* BNum → BNum → Bool *)
6 let blk_leq = (* ... *)
7
8 contract Crowdfunding
9 (* Immutable parameters *)
10 (owner : ByStr20, max_block : BNum, goal : Uint128)
11 (* Mutable fields *)
12 field backers : Map ByStr20 Uint128 = Emp ByStr20 Uint128
13 field funded : Bool = False
14 (* Transitions *)
15 transition Donate (sender : ByStr20, amount : Uint128)
16 transition GetFunds (sender : ByStr20, amount : Uint128)
17 transition ClaimBack (sender : ByStr20, amount : Uint128)
Contract Structure

```haskell
library Crowdfunding
(* Map ByStr20 Uint128 → ByStr20 → Uint128 → *)
(* Option (Map ByStr20 Uint128) *)
let check_update = (* ... *)
(* BNum → BNum → Bool *)
let blk_leq = (* ... *)

contract Crowdfunding
(* Immutable parameters *)
(owner : ByStr20, max_block : BNum, goal : Uint128)
(* Mutable fields *)
field backers : Map ByStr20 Uint128 = Emp ByStr20 Uint128
field funded : Bool = False
(* Transitions *)
transition Donate (sender : ByStr20, amount : Uint128)
transition GetFunds (sender : ByStr20, amount : Uint128)
transition ClaimBack (sender : ByStr20, amount : Uint128)
```

Fig. 2. A signature of the Crowdfunding contract.

Intermezzo 2 (External Libraries and Parity Wallet hack).

Another famous hack in Ethereum, resulting USD 146 million worth of coins becoming inaccessible (Alois 2017), was caused by non-determinism of transaction scheduling, remaining to be present even in the “transition-as-an-atomic-change” model adopted by Scilla. However, detecting those issues requires more domain-specific input from the user (Kolluri et al. 2018), and, we believe, should be addressed at a higher-level by means of a suitable domain-specific language for particular smart contract scenarios (e.g., interacting with an off-chain oracle).

This is sound, as block numbers grow monotonically, without gaps.
library Crowdfunding

(* Immutable parameters *)
(owner : ByStr20, max_block : BNum, goal : Uint128)

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

(* Transitions *)
transition Donate (sender : ByStr20, amount : Uint128)
transition GetFunds (sender : ByStr20, amount : Uint128)
transition ClaimBack (sender : ByStr20, amount : Uint128)

Fig. 2. A signature of the Crowdfunding contract.

Intermezzo 2 (External Libraries and Parity Wallet hack).
Another famous hack in Ethereum, resulting USD 146 million worth of coins becoming inaccessible (Alois 2017), was caused by non-determinism of transaction scheduling, e.g. caused by non-determinism of transaction scheduling, e.g. , and, we believe, should be addressed at a higher-level by means of a suitable domain-specific language for particular smart contract scenarios (e.g., interacting with an off-chain oracle).

This is sound, as block numbers grow monotonically, without gaps.
Contract Structure

1 library Crowdfunding
2 (* Map ByStr20 Uint128 → ByStr20 → Uint128 → *)
3 (* Option (Map ByStr20 Uint128) → *)
4 let check_update = (* ... *)
5 (* BNum → BNum → Bool *)
6 let blk_leq = (* ... *)

7 contract Crowdfunding
8 (* Immutable parameters *)
9 (owner : ByStr20, max_block : BNum, goal : Uint128)
10 (* Mutable fields *)
11 field backers : Map ByStr20 Uint128 = Emp ByStr20 Uint128
12 field funded : Bool = False
13 (* Transitions *)
14 transition Donate (sender : ByStr20, amount : Uint128)
15 transition GetFunds (sender : ByStr20, amount : Uint128)
16 transition ClaimBack (sender : ByStr20, amount : Uint128)
Contract Structure

1 library Crowdfunding
2 (* Map ByStr20 Uint128 → ByStr20 → Uint128 → *)
3 (* Option (Map ByStr20 Uint128) *)
4 let check_update = (* ... *)
5 (* BNum → BNum → Bool *)
6 let blk_leq = (* ... *)
7
8 contract Crowdfunding
9 (* Immutable parameters *)
10 (owner : ByStr20, max_block : BNum, goal : Uint128)
11 (* Mutable fields *)
12 field backers : Map ByStr20 Uint128 = Emp ByStr20 Uint128
13 field funded : Bool = False
14 (* Transitions *)
15 transition Donate (sender : ByStr20, amount : Uint128)
16 transition GetFunds (sender : ByStr20, amount : Uint128)
17 transition ClaimBack (sender : ByStr20, amount : Uint128)
Contract Structure

```plaintext
library Crowdfunding
  (* Map ByStr20 Uint128 → ByStr20 → Uint128 → *)
  (* Option (Map ByStr20 Uint128) *)
  let check_update = (* ... *)
  (* BNum → BNum → Bool *)
  let blk_leq = (* ... *)

contract Crowdfunding
  (* Immutable parameters *)
  (owner : ByStr20, max_block : BNum, goal : Uint128)
  (* Mutable fields *)
  field backers : Map ByStr20 Uint128 = Emp ByStr20 Uint128
  field funded : Bool = False
  (* Transitions *)
  transition Donate (sender : ByStr20, amount : Uint128)
  transition GetFunds (sender : ByStr20, amount : Uint128)
  transition ClaimBack (sender : ByStr20, amount : Uint128)
```
transition Donate (sender: ByStr20, amount: Uint128)
    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: ByStr20, amount: Uint128)
 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: ByStr20, amount: Uint128)

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)
transition Donate (sender: ByStr20, amount: Uint128)
  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: ByStr20, amount: Uint128)
    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: ByStr20, amount: Uint128)

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

Explicitly accepting incoming funds
Creating and sending messages

transition  Donate (sender: ByStr20, amount: Uint128)
    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
Wishlist

- Explicit interaction: no reentrancy attacks
- Minimalistic (core interpreter ~200 LOC of OCaml)
- Explicit control of effects
  - Expressive
  - Analysis/Verification friendly
  - Predictable resource (gas) consumption
  - Reasonable performance
Wishlist

- Explicit interaction: no reentrancy attacks
- Minimalistic (core interpreter ~200 LOC of OCaml)
- Explicit control of effects (e.g., acceptance of funds)
  - Expressive
  - Analysis/Verification friendly
  - Predictable resource (gas) consumption
  - Reasonable performance
Wishlist

- Explicit interaction: no reentrancy attacks
- Minimalistic (core interpreter ~200 LOC of OCaml)
- Explicit control of effects (eg, acceptance of funds)
  - Expressive
  - Analysis/Verification friendly
  - Predictable resource (gas) consumption
  - Reasonable performance
Wishlist

- Explicit interaction: no reentrancy attacks
- Minimalistic (core interpreter ~200 LOC of OCaml)
- Explicit control of effects (eg, acceptance of funds)
- Expressive
  - Analysis/Verification friendly
  - Predictable resource (gas) consumption
  - Reasonable performance
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>
Wishlist

- Explicit interaction: no reentrancy attacks
- Minimalistic (core interpreter ~200 LOC of OCaml)
- Explicit control of effects (eg, acceptance of funds)
- Expressive
  - Analysis/Verification friendly
  - Predictable resource (gas) consumption
  - Reasonable performance
Wishlist

- Explicit interaction: no reentrancy attacks
- Minimalistic (core interpreter ~200 LOC of OCaml)
- Explicit control of effects (e.g., acceptance of funds)
- Expressive (suitable for all scenarios of interest)
  - Analysis/Verification friendly
  - Predictable resource (gas) consumption
  - Reasonable performance
Wishlist

- Explicit interaction: no reentrancy attacks
- Minimalistic (core interpreter ~200 LOC of OCaml)
- Explicit control of effects (e.g., acceptance of funds)
- Expressive (suitable for all scenarios of interest)
  - Analysis/Verification friendly
  - Predictable resource (gas) consumption
  - Reasonable performance
Wishlist

- Explicit interaction: no reentrancy attacks
- Minimalistic (core interpreter ~200 LOC of OCaml)
- Explicit control of effects (eg, acceptance of funds)
- Expressive (suitable for all scenarios of interest)
- Analysis/Verification friendly
  - Predictable resource (gas) consumption
  - Reasonable performance
Verification-Friendliness

• A framework for staged static analyses (optional)

• Two instances:
  • Gas-Usage Analysis
  • Cash-Flow Analysis
Verification-Friendliness

- A framework for **staged static analyses** (optional)

- Two instances:
  - **Gas-Usage** Analysis (resources)
  - **Cash-Flow** Analysis (data flow)
Verification-Friendliness

• A framework for staged static analyses (optional)

• Two instances:
  • Gas-Usage Analysis (resources)
  • Cash-Flow Analysis (data flow)
**Cash-Flow Analysis**

```
contract Crowdfunding
(* Immutable parameters *)
(owner : ByStr20, max_block : BNum, goal : Uint128)
(* Mutable fields *)
field backers : Map ByStr20 Uint128 = Emp ByStr20 Uint128
field funded : Bool = False
(* Transitions *)
transition Donate (sender : ByStr20, amount : Uint128)
transition GetFunds (sender : ByStr20, amount : Uint128)
transition ClaimBack (sender : ByStr20, amount : Uint128)
```

Which of those correspond to currency?
Cash-Flow Analysis

- Soundly infers what fields represent money
- Based on simple abstract interpretation
- Takes user annotations for custom tokens

Lattice of Cash Tags

\[
\begin{align*}
\tau &::= \text{Money} | \text{NotMoney} | \text{Map } \tau | t \overline{\tau} | \top | \bot \\
t &::= \text{Option} | \text{Pair} | \text{List} | \ldots
\end{align*}
\]

(maps) \quad \text{Map } \tau \sqsubseteq \text{Map } \tau' \iff \tau \sqsubseteq \tau'

(algebraic types) \quad t \overline{\tau} \sqsubseteq t' \overline{\tau'} \iff t = t' \text{ and } \tau_i \sqsubseteq \tau'_i \text{ for all } i

(bottom) \quad \bot \sqsubseteq \tau \quad \text{for all } \tau

(top) \quad \tau \sqsubseteq \top \quad \text{for all } \tau
Cash-Flow Analysis

- Soundly infers what fields represent money
- Based on simple abstract interpretation
- Takes user annotations for custom tokens

```solidity
contract Crowdfunding

(* Immutable parameters *)
( owner : ByStr20, max_block : BNum, goal : Uint128 )

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

(* Transitions *)
transition Donate ( sender : ByStr20, amount : Uint128 )
transition GetFunds ( sender : ByStr20, amount : Uint128 )
transition ClaimBack ( sender : ByStr20, amount : Uint128 )
```
### Cash-Flow Analysis

<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>

15 According to https://etherscan.io/tokens, roughly 16% of active smart contracts on Ethereum are ERC20. According to https://dappradar.com/charts, games, gambling, and collectibles (ERC721) are the most used contracts on Ethereum. 16 A fungible token type is similar to traditional currencies where coins are indistinguishable and thus are freely interchangeable (if they have the same denomination). Conversely, a non-fungible token type is more similar to collector's items, e.g., paintings, where the tokens are distinguishable, and may therefore not be interchangeable. 
# Cash-Flow Analysis

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

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

*non-native tokens*
Cash-Flow Analysis

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

Notes:

- LOC: Lines of Code
- #Lib: Number of Library Functions
- #Trans: Number of Transactions
- $-Flow: Cash-Flow Analysis

- ✓: Fungible tokens
- ✓⊥: Non-fungible tokens
- ✓*: Unsure due to high complexity

Non-fungible tokens are further explained in the text.
Wishlist

- Explicit interaction: no reentrancy attacks
- Minimalistic (core interpreter ~200 LOC of OCaml)
- Explicit control of effects (eg, acceptance of funds)
- Expressive (suitable for all scenarios of interest)
- Analysis/Verification friendly
  - Predictable resource (gas) consumption
  - Reasonable performance
Wishlist

- Explicit interaction: no reentrancy attacks
- Minimalistic (core interpreter ~200 LOC of OCaml)
- Explicit control of effects (eg, acceptance of funds)
- Expressive (suitable for all scenarios of interest)
- Analysis/Verification friendly
  - Predictable resource (gas) consumption
  - Reasonable performance
Wishlist

- Explicit interaction: no reentrancy attacks
- Minimalistic (core interpreter ~200 LOC of OCaml)
- Explicit control of effects (eg, acceptance of funds)
- Expressive (suitable for all scenarios of interest)
- Analysis/Verification friendly
- Predictable resource (gas) consumption
  - Reasonable performance
Wishlist

- Explicit interaction: no reentrancy attacks
- Minimalistic (core interpreter ~200 LOC of OCaml)
- Explicit control of effects (eg, acceptance of funds)
- Expressive (suitable for all scenarios of interest)
- Analysis/Verification friendly
- Predictable resource (gas) consumption
  - Reasonable performance
Wishlist

- Explicit interaction: no reentrancy attacks
- Minimalistic (core interpreter ~200 LOC of OCaml)
- Explicit control of effects (eg, acceptance of funds)
- Expressive (suitable for all scenarios of interest)
- Analysis/Verification friendly
- Predictable resource (gas) consumption
- Reasonable performance
Relative Code Size

### 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>

### Fig. 12. Runtime and size statistics on some representative smart contracts.

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.

17 The artefact containing the benchmarks is available on GitHub: [https://github.com/ilyasergey/scilla-benchmarks](https://github.com/ilyasergey/scilla-benchmarks).

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

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.
Wishlist

- Explicit interaction: no reentrancy attacks
- Minimalistic (core interpreter ~200 LOC of OCaml)
- Explicit control of effects (eg, acceptance of funds)
- Expressive (suitable for all scenarios of interest)
- Analysis/Verification friendly
- Predictable resource (gas) consumption
- Reasonable performance
Wishlist

- Explicit interaction: no reentrancy attacks
- Minimalistic (core interpreter ~200 LOC of OCaml)
- Explicit control of effects (eg, acceptance of funds)
- Expressive (suitable for all scenarios of interest)
- Analysis/Verification friendly
- Predictable resource (gas) consumption
- Reasonable performance (in a ballpark of EVM)
Wishlist

- Explicit interaction: no reentrancy attacks
- Minimalistic (core interpreter ~200 LOC of OCaml)
- Explicit control of effects (eg, acceptance of funds)
- Expressive (suitable for all scenarios of interest)
- Analysis/Verification friendly
- Predictable resource (gas) consumption
- Reasonable performance (in a ballpark of EVM)
Wishlist

- Explicit interaction: no reentrancy attacks
- Minimalistic (core interpreter ~200 LOC of OCaml)
- Explicit control of effects (e.g., acceptance of funds)
- Expressive (suitable for all scenarios of interest)
- Analysis/Verification friendly
- Predictable resource (gas) consumption
- Reasonable performance (in a ballpark of EVM)
Adoption

- Scilla launched on Zilliqa test-net in June 2018, on main-net 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
1 let new_mig =
2 fun (msg : Message) ->
3   let nil_mig = Nil (Message) in
4   Core (Message) msg nil_mig
5
6 Library Crowdfunding

10 let check_updates =
11 fun (bs : Map (ByStr20) Unit112) ->
12 fun (_sender : ByStr20) ->
13 fun (_amount : Int112) ->
14 let c = builtin contains hl_sender in
15 match c with
16 | False ->
17 | let hst1 = builtin put hst_sender_amount in
18 | Some (Map ByStr20 Unit112) hst1
19 | True ->
20 | More (Map ByStr20 Unit112) hst1
21 | end
22
23 let k1_leq =
24 fun (H1KL : B1211) ->
25 fun (H1KL : B1212) ->
26 let hl1 = builtin bit h1k1 h1k2 in
27 let h2 = builtin ac h1k1 h1k2 in
28 and h1 and h2
Contract

Balance 193.35 ZIL

Transactions 40

Contract Creation zil1fxx... at dab4e4878c0d93abcaa...

```
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

- Ethereum: 21.1%
- Bitcoin: 20.1%
- Qtum: 12.5%
- Dash: 11.2%
- Zilliqa: 4.7%
- Litecoin: 3.9%
- Monero: 3.2%
- Bitcoin Cash: 2.8%
- Algorand: 2.5%
- TRON: 2.5%
- Ripple: 2.3%
- Harmony: 1.2%
- Ethereum Classic: 1.0%

William Mougayar © Sept 22 2019 v1
To Take Away

• Adopting a foundational calculus is a great way to keep a new language *minimalistic* and *expressive*.

• Lots of ideas from PL research can be reused with very low overhead on implementation and adoption.

• Yet the language will be forced to *grow* and *change*.

• It pays off to build an enthusiastic developer community: more feedback — more *informed design choices*.

scilla-lang.org

Thanks!