Implementing and Mechanically Verifying Smart Contracts

Ilya Sergey

ilyasergey.net
Smart Contracts

• *Stateful mutable* objects replicated via a (Byzantine) consensus protocol
• State typically involves a stored amount of *funds/currency*
• One or more entry points: invoked *reactively* by a client *transaction*

• Main usages:
  • crowdfunding and ICO
  • multi-party accounting
  • voting and arbitration
  • puzzle-solving games with distribution of rewards

• Supporting platforms: *Ethereum, Tezos, Zilliqa, EOS, ...*
contract Accounting {
    /* Define contract fields */
    address owner;
    mapping (address => uint) assets;

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

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

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

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

  function stealMoney() {
    if (msg.sender == owner) { owner.send(this.balance) }
  }
}
Misconceptions about Smart Contracts

Deployed in a low-level language

Uniform compilation target

Must be Turing-complete

Run arbitrary computations

Code is law

What else if not the code?
Misconceptions about Smart Contracts

Deployed in a low-level language

Must be Turing-complete

Code is law

Infeasible audit and verification

DoS attacks, cost semantics, exploits

Cannot be amended once deployed
What about High-Level Languages?

```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";
    }
}
```

Ethereum's **Solidity**

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

contract accounting {
  /* define contract fields */
  address owner;
  mapping (address => uint) assets;

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

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

ethereum's solidity
• javascript-like syntax
• calling a function = sending funds
• general recursion and loops
• reflection, dynamic contract creation
• lots of implicit conventions
• no formal semantics
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;
  }
}
```

Can reenter and withdraw again
# Smart Contracts in a Nutshell

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>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
Smart Contract Intermediate-Level Language

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

Primitive type $P \ ::= \ Int$ \hspace{1cm} integer
String \hspace{1cm} string
Hash \hspace{1cm} hash
BNum \hspace{1cm} block number
Address \hspace{1cm} account address

Type $T, S \ ::= \ P$ \hspace{1cm} primitive type
\hspace{1cm} map
\hspace{1cm} message
\hspace{1cm} value function
\hspace{1cm} instantiated data type
\hspace{1cm} type variable
\hspace{1cm} polymorphic function
Expressions (pure)

<table>
<thead>
<tr>
<th>Expression</th>
<th>e ::= f</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>let x : T = f in e</td>
</tr>
<tr>
<td>Simple expression</td>
<td>f ::= l</td>
</tr>
<tr>
<td></td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>{ (entry) }</td>
</tr>
<tr>
<td></td>
<td>fun (x : T) =&gt; e</td>
</tr>
<tr>
<td></td>
<td>builtin b (x)</td>
</tr>
<tr>
<td></td>
<td>x (x)</td>
</tr>
<tr>
<td></td>
<td>tfun α =&gt; e</td>
</tr>
<tr>
<td></td>
<td>@x T</td>
</tr>
<tr>
<td></td>
<td>C ( (T) ) (x)</td>
</tr>
<tr>
<td></td>
<td>match x with (</td>
</tr>
<tr>
<td>Selector</td>
<td>sel ::= pat =&gt; e</td>
</tr>
<tr>
<td>Pattern</td>
<td>pat ::= x</td>
</tr>
<tr>
<td></td>
<td>C (pat)</td>
</tr>
<tr>
<td></td>
<td>(pat)</td>
</tr>
<tr>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Message entry</td>
<td>entry ::= b : x</td>
</tr>
<tr>
<td>Name</td>
<td>b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type conversions</td>
<td>Int to nat conversion</td>
<td></td>
</tr>
<tr>
<td>Built-in operations and conversions on primitive data types.</td>
<td>Figure 3.</td>
<td></td>
</tr>
</tbody>
</table>
Structural Recursion in Scilla

Natural numbers (not Ints!)

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

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

```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 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

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) =>
5.       match res with
6.       | And x y => let z = builtin add x y in
7.       | And {Int Int} z x
8.       end
9.     in
10.    let zero = 0 in
11.    let one = 1 in
12.    let init_val = And {Int Int} one zero in
13.    let res = iter_nat init_val iter_fun n in
14.    fst res

Value for 0: (1, 0)
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 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
```

(Ilya: Emphasize the sending of several messages)
Example: Fibonacci Numbers

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

The result of iteration is a pair of integers
Example: Fibonacci Numbers

```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
```
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
  let zero = 0 in
  let one = 1 in
  let init_val = And {Int Int} one zero in
  let res = iter_nat init_val iter_fun n in
  fst res
```

return the first component of the result pair
Why Structural Recursion?

• Pros:
  • *All programs terminate*
  • Number of operations can be computed *statically* as a function of *input size*

• Cons:
  • Some functions cannot be implemented efficiently (e.g., QuickSort)
  • Cannot implement *Ackerman function* :

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

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

read from mutable field
store to a field
assign a pure expression
pattern matching and branching
read from blockchain state
accept incoming payment
send list of messages
Statement Semantics

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

**BlockchainState**

Immutable global data (block number etc.)

**Configuration**

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

- Immutable bindings
- Contract's own funds
- Messages to be sent
- Funds sent to contract
- Mutable fields
Global Execution Model

Account X
Global Execution Model

Account X → Contract C

m_1

Contract D → Contract C

m_2

Contract C → Contract E

m_3

m_4

m_5

Account Z → Contract C

m_6

Account Y → Contract C
Global 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}'

Fixed MAX length of call sequence

Final contract states
Global Execution Model

Conf_0

Conf

Conf

Conf

Conf

Conf

Conf

m_1

m_6

Conf''_C
Putting it All Together

• Scilla contracts are (infinite) *State-Transition Systems*

• Interaction *between* contracts via sending/receiving *messages*

• Messages trigger (effectful) *transitions* (sequences of *statements*)

• A contract can *send messages* to other contracts via *send* statement

• Most computations are done via *pure expressions*, no storable closures

• Contract's state is *immutable parameters*, *mutable fields*, *balance*
Contract Structure

- Library of pure functions
- Immutable parameters
- Mutable fields

Transition 1
...
Transition N
Working Example: *Crowdfunding* contract

- **Parameters**: campaign's *owner*, deadline (max block), funding *goal*
- **Fields**: registry of backers, "*campaign-complete*" boolean flag
- **Transitions**:
  - **Donate** money (when the campaign is active)
  - **Get funds** (as an owner, after the deadline, if the goal is met)
  - **Reclaim** donation (after the deadline, if the goal is not met)
transition Donate (sender: Address, amount: Int)
    blk <- & BLOCKNUMBER;
    in_time = blk_leq blk max_block;
match in_time with
| True  =>
    bs <- backers;
    res = check_update bs sender amount;
match res with
| None =>
    msg = {tag : Main; to : sender; amount : 0; code : already_backed};
    msgs = one_msg msg;
    send msgs
| Some bs1 =>
    backers := bs1;
    accept;
    msg = {tag : Main; to : sender; amount : 0; code : accepted_code};
    msgs = one_msg msg;
    send msgs
    end
| False =>
    msg = {tag : Main; to : sender; amount : 0; code : missed_dealine};
    msgs = one_msg msg;
    send msgs
    end
end
transition Donate (sender: Address, amount: Int)

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

Numeric code to inform the recipient
Demo
Verifying Scilla Contracts

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

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

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

- “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

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

**Definition** since\_as\_long

(\(P : \text{conf } \rightarrow \text{Prop}\))

(\(Q : \text{conf } \rightarrow \text{conf } \rightarrow \text{Prop}\))

(\(R : \text{bstate } \times \text{message } \rightarrow \text{Prop}\) :=

\(∀ \text{ sc conf conf',}

P \text{ st } \rightarrow

(\text{conf } \leftrightarrow \text{conf'} \text{ sc}) \land (\forall b, b \in \text{sc} \rightarrow R 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 : address) (d : amount) conf :=
conf.backers(b) == d.

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

**Lemma** donation_preserved (b : address) (d : amount):
since_as_long (donated b d) (fun c c' => donated b d c')
(no_claims_from b).

*b*’s records are preserved by the contract.
Demo
Misconceptions, revisited

Need a low-level language
Need a language easy to reason about

Must be Turing-complete
Primitive recursion suffices in most cases

Code is law
Code should abide by a specification
What’s next?

• Certified interpreter for Scilla contracts

• Compilation into an efficient back-end (LLVM, WASM)

• Certifications for *Proof-Carrying Code* (storable on a blockchain)

• *Automated Model Checking* smart contract properties

• PL support for *sharded contract executions*
To Take Away

- Formal verification of *functional* and *temporal* properties of smart contracts requires a language with a clear separation of concerns.

- Scilla: is a Smart Contract Intermediate-Level Language that provides it:
  - **Small**: builds on the polymorphic lambda-calculus with extensions.
  - **Principled**: separates computations, effects, and communication.
  - **Verifiable**: formal semantics and methodology for machine-assisted reasoning.

Thanks!
• Do you want to work on *formal proofs* of correctness for *practical distributed systems* and *smart contracts* in **Coq**?

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

• Also, *postdoc positions* at Yale-NUS College are available starting **early 2019**.

• Get in touch with questions about topics and positions
  • Check [ilyaserey.net](http://ilyaserey.net) for my contact details.