Is Solidity solid enough?

Silvia Crafa **Matteo Di Pirro** **Elena Zucca**

Università di Padova
Italy

Università di Genova
Italy

WTSC 2019 *22 February 2019 - St Kitts & Nevis*

Trusted Smart Contracts

Safety properties of Ethereum's Smart Contracts can be verified either

- ***dynamically:*** during program execution,
  - but "test"-blockchains must be used since incorrect executions over Ethereum cause Ether to be lost

- ***statically:*** specific safety properties are checked before running the code
  - static ***analysis*** *of the* ***bytecode*** by means of tools working at the level of Ethereum Virtual Machine
  - static ***analysis*** *of the* ***source code*** directly written by smart contracts' programmers

Trusted Smart Contracts

Safety properties of Ethereum's Sma

- ***dynamically:*** during program exe
  - but "test"-blockchains must be
    Ethereum cause Ether to be lost.

- ***statically:*** specific safety properties are checked before running the code
  - static ***analysis*** *of the* ***byte code*** by means of tools working at the level of Ethereum Virtual Machine
  - static ***analysis*** *of the* ***source code*** directly written by smart contracts' programmers

we take this approach, focusing on

Solidity programming language

(the most widely used smart contract language in Ethereum ecosystem)

Trusted Solidity Smart Contracts

We perform static analysis of the Solidity source code, **so to**:

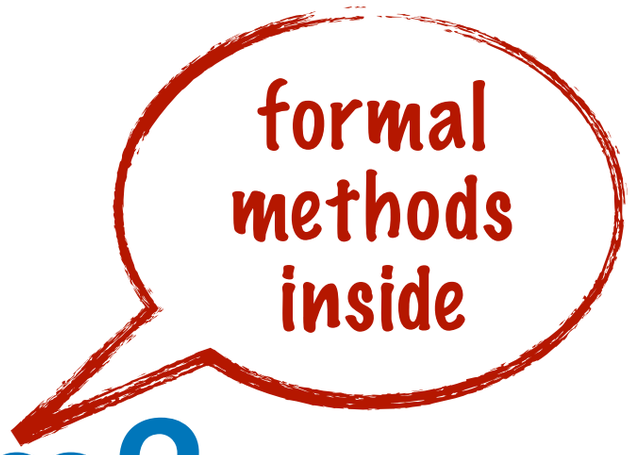
* ***statically*** ***rule out harmful programming patterns*** appearing in the source code

* ***support a safer programming discipline***, to write safer programs from the beginning

* we use ***types as a static analysis tool***, because
  - Solidity is a typed language, and it is claimed to be “type safe”
  - Solidity programmers commonly use the compiler to check type errors in the source code
  - we want to enhance the use of *compiler and a convenient building tool*

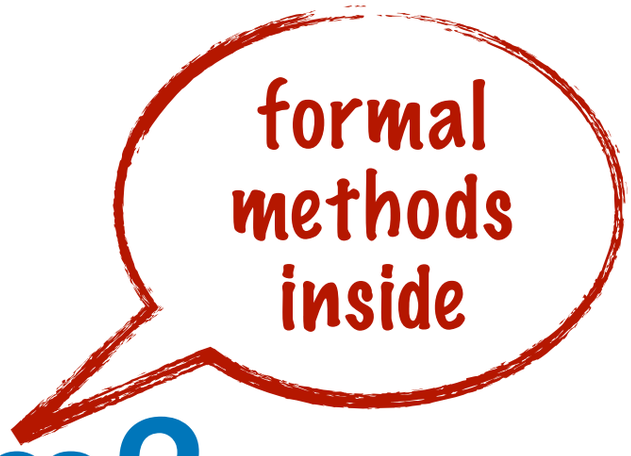
…these are the goals,

how do we reach them?



…these are the goals,

how do we reach them?



1. **We formally study the core of the Solidity Language** (*Featherweight Solidity-FS*)
   - thus giving a precise account of the smart contracts behaviour

2. **We formally study the type system of FS**
   - thus studying the soundness of the Solidity compiler

3. **We propose a refinement of the Solidity/FS type system** that statically captures more runtime errors and is **retro-compatible** with original Solidity code
   - thus it is possible for contracts written in the extended safer language to interact with already deployed smart contracts.

1. Featherweight Solidity

Featherweight Solidity (FS) **formally captures the core of Solidity**:

and **precisely** (*i.e. in a mechanically analysable way*) **describes the subtle behaviours** of many smart contract programs

A taste of Featherweight Solidity….

A Bank Smart Contract

Solidity code

```
contract Bank {
  mapping (address => uint) amounts;

  function withdraw(uint n) {
     require(amounts[msg.sender] >= n);
     amounts[msg.sender] -= n;
     msg.sender.transfer(n);
  }

  function deposit() payable {
     amounts[msg.sender] += msg.value;
  }
}
```

A Bank Smart Contract

Solidity code

```
contract Bank {
  mapping (address => uint) amounts;

  function withdraw(uint n) {
     require(amounts[msg.sender] >= n);
     amounts[msg.sender] -= n;
     msg.sender.transfer(n);
  }

  function deposit() payable {
     amounts[msg.sender] += msg.value;
  }
}
```

state variable that records the amounts of money deposited by clients (either EOAs or smart contracts) indexed by their Ethereum addresses.

if the caller has not deposited enough money, a **`revert`** exception is thrown and the transaction is rolled-back,

otherwise `n` Wei are transferred from the balance of **`this`** to the balance of the caller (**`msg.sender`**), moreover the caller's ***fallback*** function is called, if not present a **`revert`** is thrown and the transaction is aborted

`b.deposit().value(50)` binds **`msg.value`** to 50 and 50 Wei are transferred from the balance of the caller (**`msg.sender`**) to the balance of the Bank instance **`b.`**
If the caller has not got enough money a **`revert`** is thrown

A Bank Smart Contract

Featherweigh Solidity code

```
contract Bank {
  mapping (address => uint) amounts;

  unit withdraw(uint n) {
    return if this.amounts[msg.sender] >= n);
           then this.amounts[msg.sender] -= n;
                msg.sender.transfer(n); u
           else revert
  }

  unit deposit(){
   return this.amounts[msg.sender]+=msg.value; u
  }
}
```

still quite similar to Solidity code

look at the paper for the precise FS **syntax**

- no function markers (all `external` and `payable`)
- every function returns a value, possibly `unit`
- `this` and `revert` explicitly written

A Bank Transaction

Featherweigh Solidity code

```
contract Bank {
  mapping … amounts;

  unit withdraw(uint n) {
   …
   msg.sender.transfer(n);
  }

  unit deposit(){
   …
  }
}
```

look at the paper for the precise FS **semantics**

a user contract that defines the ***fallback*** function

```
contract A {
    ...
    unit fb() { return … }
 }
```

```
Bank('0x84b').deposit.value(50).sender('0xu7e')()
```

denotes a **transaction**
issued by an instance of contract **A** at address '0xu7e'
to interact with
an instance of the **Bank** contract stored at address '0x84b'

A Bank Transaction

Featherweigh Solidity code

```
contract Bank {
  mapping … amounts;

  unit withdraw(uint n) {
   …
   msg.sender.transfer(n);
  }

  unit deposit(){
   …
  }
}
```

a user contract **without** ***fallback*** function

```
contract A {
    ...
   unit fb() { return … }
 }
```

```
Bank('0x84b').deposit.value(50).sender('0xu7e')()
```

```
Bank('0x84b').withdraw.value(0).sender('0xu7e')(10)
```

… ⟶ `'0xu7e'.transfer(10)` ⟶ **revert**

- the second transaction evolves to a call to **`transfer`** on the contract **`A`**, which throws a **`revert`** since there is no fallback function in the code of **`A`**
- the code of deployed contracts **cannot be modified** anymore!
- the 50 Wei deposited by contract A are **locked in the blockchain!**

A Bank Transaction

Featherweigh Solidity code

```
contract Bank {
  mapping … amounts;

  unit withdraw(uin
   …
   msg.sender.transi
  }

  unit deposit(){
   …
  }
}
```

Solidity **compiler does not complain** with this transaction!

a user contract **without *fallback*** function

```
contract A {
    ...
   unit fb() { return … }
 }
```

```
Bank('0x84b').deposit.value(50).sender('0xu7e')()
```

```
Bank('0x84b').withdraw.value(0).sender('0xu7e')(10)
```

… ⟶ `'0xu7e'.transfer(10)` ⟶ **`revert`**

- the second transaction evolves to a call to **`transfer`** on the contract **`A`**, which throws a **`revert`** since there is no fallback function in the code of **`A`**
- the code of deployed contracts **cannot be modified** anymore!
- the 50 Wei deposited by contract A are **locked in the blockchain!**

A Bank Transaction

Featherweigh Solidity code

```
contract Bank {
  mapping … amounts;

  unit withdraw(uin
   …
   msg.sender.transi
  }

  unit deposit(){
   …
  }
}
```

Solidity **compiler does not complain** with this transaction!

a user contract **without *fallback*** function

```
contract A {
    ...
   unit fb() { return … }
 }
```

```
Bank('0x84b').deposit.value(50).sender('0xu7e')()
```

```
Bank('0x84b').withdraw.value(0).sender('0xu7e')(10)
```

moreover, the compiler does not check **whether it is really the address of a `Bank` contract**, if not, it silently goes to revert!

2. The FS type system

We formally study the FS type system

- it is the foundational core of Solidity compiler
- the **type soundness theorem of FS clarifies** (by precisely stating) **the Solidity claim to be a "type-safe language":**
  - Solidity/FS static typing **only prevents stuck expressions** but **not runtime type errors**, such as **accesses to a non existing function** (such as fallback) or state variable

look at the paper for the precise FS **type system and Soundness Theorem**

3. Improve the power of types

The problem comes from Solidity type `address`:

- it is **an untyped way to access contract references**, thus there is no static guarantee about safe accesses to the contract's members
- much as `void *` pointers in C, which are **flexible** but **error-prone**

3. Improve the power of types

The problem comes from Solidity type `address`:

- it is **an untyped way to access contract references**, thus there is no static guarantee about safe accesses to the contract's members
- much as `void *` pointers in C, which are **flexible** but **error-prone**

Solution:

1. **refine address types** into `address<C>` to constrain the type of the contract the address may refer to
2. **refine function signatures** to **constrain the (address) type of the caller:**
   `function f(T x) <C>` defines a function that **can be called only by contracts of type** (lower than) `C`

look at the paper for the **refined typing**

A type safer Bank contract

Solidity+ code

```
contract Bank {
  mapping (address<Top_fb> => uint) amounts;
  function withdraw(uint n) <Top_fb> { … }
  function deposit() payable <Top_fb> { … }
}
```

they must be addresses of contracts that contain a fallback function
(i.e. any type C such that **address<C> <: address<Top_fb>**)

the caller of these functions **must be able to accept money back**

the **compiler now prevents unsafe** transactions!

…programmers have to annotate interfaces with additional types

A type safer Bank contract

according to Solidity language style,

we introduce useful **function modifiers** and syntactic sugar

to denote the (super)type of *contracts that are able to accept money back*

```
contract Bank {
  mapping (payableaddress => uint) amounts;

  function withdraw(uint n) payback { … }

  function deposit() payable payback { … }
}
```

Solidity+ code

the **compiler now prevents unsafe** transactions!

function modifiers are verbose but **support a safer programming discipline**

Conclusions

1. **We formally study FS, modelling the core of Solidity,** so to precisely describe the subtle behaviour of many smart contract programs
   - FS **unleashes many well-known static analysis techniques**, that require an underlying formal semantics to operate
   - FS **highlights the connections** and the differences **with OOLs** and their rich (type) theory. FS is inspired by ***Featherweight Java*** [Igarashi,Pierce,Wadler,2002]

2. The FS type system provides **a foundation of Solidity compiler's soundness**

3. **We propose a refinement of the Solidity/FS type system** that statically captures more runtime errors and is **retro-compatible** with original Solidity code, allowing new, safer, contracts to interact with already deployed smart contracts.

thank you

and sorry for not being here,

for questions … email us

Silvia, Matteo, Elena