Smart Contracts

Danil Annenkov Bas Spitters

Aarhus University

April 5, 2019

Outline

Smart contracts

- A concept proposed by Nick Szabo in 90s.
- (Wikipedia) A smart contract is a computer protocol intended to digitally facilitate, verify, or enforce the negotiation or performance of a contract.
- Usually thought as self-enforcing, self-executing entities.

Smart contracts

- A concept proposed by Nick Szabo in 90s.
- (Wikipedia) A smart contract is a computer protocol intended to digitally facilitate, verify, or enforce the negotiation or performance of a contract.
- Usually thought as self-enforcing, self-executing entities.

This is not what "smart contracts" on blockchains are!

Smart contracts are neither

At least currently we have:

- Smart contracts are programs in a general purpose language.
- Connection to the legal contracts in not clear.
- Smart contracts mix together specification and execution.
- Can go terribly wrong.

Fritz Henglein. Smart contracts are neither.
Cyber Security, Privacy and Blockchain High Tech Summit, DTU, 2017

Smart contracts: evolution

- First generation: bitcoin script and alike.
- Second generation: Ethereum EVM and Solidity.
- Third generation: functional languages + limited inter-contract communication patterns.

Bitcoin script

- Forth-like stack-based language.
- Not Turing complete (no recursion or loops).
- Used to validate Bitcoin transactions.
- Might be used to create simple smart contracts, e.g.
  - multi-signature transactions;
  - timelocked transactions — can be spend only after specified time.
- No inter-contract communication.

Ethereum and Solidity

- Solidity is a high level java/javascript-like imperative language.
- Compiles to EVM byte-code.
- Each contract has state, which can be modified during the execution of any of contract's methods.
- Contracts can interact with other contracts by calling methods and sending money.
- Calls can happen in any point of the program execution.

What is Gas?

- A measure of computational efforts.
- Allows for decoupling transaction fee calculations from the native currency cost.
- Spent gas is a transaction fee that miners get as a reward.

What can go wrong here?

```
mapping (address => uint) private userBalances;

function withdrawBalance() public {
  // lookup user balance in the map
  uint amountToWithdraw = userBalances[msg.sender];

  if (amountToWithdraw > 0) {
  // send ether to the sender's address
  // (can trigger code execution on the sender's side)
  require(msg.sender.call.value(amountToWithdraw)());

  // reset user's balance to zero
  userBalances[msg.sender] = 0;
  }
}
```

Is Solidity really solid?

Plenty of vulnerabilities have been found:

- Adrian Manning. *Solidity Security: Comprehensive list of known attack vectors and common anti-patterns*[1]
  16 Solidity Hacks/Vulnerabilities
- Luu et al. *Making Smart Contracts Smarter*[2].
  19366 contracts analised, 8833 of them have vulnerabilities.
- Ilya Sergey, Aquinas Hobor. A Concurrent Perspective on Smart Contracts[3]
  Multiple issues related to (non-obvious) concurrent behaviour

[1] https://blog.sigmaprime.io/solidity-security.html
[2] https://eprint.iacr.org/2016/633.pdf
[3] https://arxiv.org/pdf/1702.05511.pdf

Towards safer smart contract languages

Why designing safe smart contract languages is crucially important?
At least, because:

- Many contract implementers with different backgrounds ("coding" is becoming a mass culture).
- Once deployed, contract code cannot be changed.
- Contract execution is irreversible ("Code is Law").
- Flaws in a smart contract may result in huge financial losses (infamous DAO smart contract on Ethereum).

Towards safer smart contract languages

Why designing safe smart contract languages is crucially important?
At least, because:

- Many contract implementers with different backgrounds ("coding" is becoming a mass culture).
- Once deployed, contract code cannot be changed.
- Contract execution is irreversible ("Code is Law").
- Flaws in a smart contract may result in huge financial losses (infamous DAO smart contract on Ethereum).

Safe languages should make shooting yourself in the foot if not impossible, but at least hard!

A functional perspective on smart contracts

How we can address the issues? Functional languages to the rescue!

- Based on variants of typed λ-calculi.
- Well-studied **formal semantics**.
- Well-suited for reasoning.
- Proof assistants are based on typed λ-calculi as well!

Semantics matters

Why do we care about formal semantics?

- Meta-theory of a language:
  - type soundness "well-typed programs can't go wrong";
  - termination;
  - compiler correctness;
- Program correctness.

Meta-theory of polymorphic λ-calculus (a.k.a System F) is well developed and forms a solid basis of many functional languages.

Functional core, imperative shell

It's all is good, **but**

- We cannot get rid of stateful computations completely — blockchains are inherently stateful.
- However, we can limit ways of modifying the state.
- Contracts are pure functions transforming the state:

```
contract : state * parameters -> state * operation list
```

Functional core, imperative shell

It's all is good, **but**

- We cannot get rid of stateful computations completely — blockchains are inherently stateful.
- However, we can limit ways of modifying the state.
- Contracts are pure functions transforming the state:

```
contract : state * parameters -> state * operation list
```

Examples of languages with the functional "core".

- Simplicity — https://blockstream.com/simplicity.pdf
- Plutus — https://testnet.iohkdev.io/plutus/
- **Liquidity/Michelson** — http://www.liquidity-lang.org/
- Scilla — https://scilla-lang.org/
- Oak — https://www.concordium.com/technology/

Outline

Liquidity

- OCaml-like functional language for the Tezos platform.
- Supports OCaml syntax and ReasonML (JavaScript-like) syntax.
- Compiles to Michelson — a stack-based functional language.
- Semantics is given by the compilation schema.
- Easy to experiment with contracts using the online-editor.
  http://www.liquidity-lang.org/edit/

Liquidity: inter-contract communication model

- **Unlike** Solidity, it is **not possible** to call contracts and modify the contract state in the course of the contract execution.
- Contract consists of **entry points**: functions
  `entry_point_N : param * storage -> operation list * storage`
- `storage` — a (user-defined) type of internal state of a contract.
- This makes contracts pure functions taking current state as input and producing new state and a list of operations.

Liquidity: program structure

```
[%%version 1.0]

<... local declarations ...>

type storage = TYPE

let%init storage
    (x : TYPE)
    (y : TYPE)
    ... =
    BODY

let%entry entrypoint2
    (p2 : TYPE)
    (s2 : TYPE) =
    BODY
let%entry main
    (parameter : TYPE)
    (storage : TYPE) =
    BODY
 ...
```

Liquidity: an example

```
(* A counter contract *)
[%%version 1.0]

type storage = int

let%init storage = 0

(* state transforming function, applied to a concrete
   increment has type [storage -> storage] *)
let inc  (increment : int) : storage -> storage =
  fun (s : storage) -> s + increment

(* entry point takes two arguments *)
let%entry counter_entry (p : int) (s : storage) =
  ( [], inc p s)
```

Liquidity: data types

- Usual primitive types: `unit`, `bool`, `nat`, `int`, `string` ...
- Blockchain-specific primitive types: `tez`, `key`, `signature`, `operation`
- Container types: `'a list`, `'a set`, `'key 'val map`
- Algebraic data types (not recursive):
  - predefined: `type 'a option = None | Some of 'a`
  - custom `type msg = Stop | Start | IncBy of nat`
- Records: `type storage = { x : string; y : int; }`
- Function types: `nat -> nat`, polymorphic functions `'a -> 'b`

Liquidity: language constructs

- Most of the usual OCaml constructs: `let`, pattern-matching `match x with ...`, anonymous functions `fun (x : nat)-> x + 2`.
- Tuples

```
let t = (x,y,z) in
let should_be_true = t.(2) = z in ...
```

- Records
  - creation and field access

```
let r = { x = "foo"; y = 3 } in
r.x
```

  - field "update" syntax

```
let r = { x = "foo"; y = 3 } in
r.x <- "bar"
```

- General recursion is supported (but execution is terminating due to gas limits).
- More details here: http://www.liquidity-lang.org/doc/index.html

Liquidity: calling other contracts

- Calls are results of the execution of your contract along with the new state.
- To create a contract call or a transfer:

```
type storage = unit

let%entry main ( to_forward : tez ) _ =
  let dest = (tz1YLtLqD1fWHthSVHPD116oYvsd4PTAHUoc :
      UnitContract.instance) in
  let op = Contract.call ~dest ~amount:to_forward () in
  [op], ()
```

- Alternatively

```
      let op = dest.main ~amount:to_forward ()
```

Liquidity: the `Current` module

- Allows to inspect many parameters of the current program execution.
- `Current.balance : unit -> tez` the balance of the current contract.
- `Current.time : unit -> timestamp` the timestamp of the block in which the transaction is included.
- `Current.amount : unit -> tez` the amount of tez transferred by the current operation
- `Current.gas : unit -> nat` amount of gas available to execute the rest of the transaction.
- `Current.sender : unit -> address` the address that initiated the current transaction
- `Current.failwith : 'a -> 'b` makes the current transaction and all its internal transactions fail.

Compiling to Michelson

- Michelson is a stack-based monomorphic functional language with simple semantics (formalised in Coq).
- Records and algebraic data types are compiled to tuples and sum types (variants).
- Michelson does not support closures, Liquidity programs are translated by lambda-lifting.
- Polymorphic functions are supported, but they will be monomorphised in Michelson.
- Michelson programs can be decompiled back to Liquidity.

Try-Liquidity

DEMO

http://www.liquidity-lang.org/edit/

Outline

Exercises

You can find the exercises, homework and some supplementary materials here

`https://github.com/annenkov/LBS`

You are welcome to send your questions and solutions to `daan@cs.au.dk`, or come by my office Turing-226.