Smart Contracts

Danil Annenkov and Bas Spitters

April 22, 2022

Outline

Smart contracts: an overview

CameLIGO

Concordium's Rust Smart Contracts

Exercises

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.

Smart contracts on the blockchain are related, but different

Smart contracts are neither (smart nor contracts)[1]

Smart contracts for blockchains:

programs in an (almost) general-purpose language deployed on a blockchain

Somewhat like stored procedures for a fancy database.

- Check conditions, change account balances and user-defined contract state.
- Code and calls to contracts *from users* are recorded in blocks.
- Can call other contracts containing possibly malicious code.
- Each node executes the calls and maintains state.
- Contracts are not self-executing: someone has to call.

[1] Comparison with declarative contract languages: Fritz Henglein. Smart contracts are neither.

Smart contracts: use cases

- auctions
- crowdfunding campaigns
- tokens
- decentralised exchanges
- ...

Smart contracts: evolution

- First generation: bitcoin script and alike.
- Second generation: Ethereum EVM and Solidity.
- Third generation: type safe languages grounded in theory.

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*[2]
  16 Solidity Hacks/Vulnerabilities
- Luu et al. *Making Smart Contracts Smarter*[3].
  19366 contracts analised, 8833 of them have vulnerabilities.
- Ilya Sergey, Aquinas Hobor. A Concurrent Perspective on Smart Contracts[4]
  Multiple issues related to (non-obvious) concurrent behaviour

---

[2] `https://blog.sigmaprime.io/solidity-security.html`
[3] https://eprint.iacr.org/2016/633.pdf
[4] 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 substantial 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 substantial financial losses (infamous DAO smart contract on Ethereum).

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

Towards safer smart contract languages, cont.

How we can address these issues?

Towards safer smart contract languages, cont.

How we can address these issues?

Use decades of research in programming languages!

- Typed λ-calculi.
  - Well-studied **formal semantics**.
  - Well-suited for reasoning.
  - Proof assistants are based on typed λ-calculi as well!
- Modern type-safe hybrid paradigm languages — Rust.
  - Functional subset.
  - Advanced type system.
  - "Core" Rust is formally studied.

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 and hybrid languages.

Functional core, imperative shell

- Some blockchains use functional programming (FP) languages.
- Very few side-effects.
- Solid theoretical foundation, easier to verify programs.

Functional core, imperative shell

- Some blockchains use functional programming (FP) languages.
- Very few side-effects.
- Solid theoretical foundation, easier to verify programs.

However

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

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

- A *scheduler*: updates the state, handles transfers and calls `operation list`.

Functional core, imperative shell

- Some blockchains use functional programming (FP) languages.
- Very few side-effects.
- Solid theoretical foundation, easier to verify programs.

However

- We cannot get rid of state — blockchains are inherently stateful.
- We can limit ways of modifying the state.
- Contracts are pure functions transforming the state:
  `contract : state * parameters -> state * operation list`
- A *scheduler*: updates the state, handles transfers and calls `operation list`.

Examples of languages with the functional “core”.

- **LIGO** (Tezos)
- **Liquidity** (Dune)
- Scilla (Zilliqa)
- Plutus (Cardano)

Hybrid-paradigm languages

- Languages with imperative features, but striving for safety.
- Feature advanced type systems similar to FP languages.
- Have clearly defined specification (formal or informal).
- Designed with verification in mind.

Hybrid-paradigm languages

- Languages with imperative features, but striving for safety.
- Feature advanced type systems similar to FP languages.
- Have clearly defined specification (formal or informal).
- Designed with verification in mind.

- **Rust** (Concordium).
- `Fe` (Ethereum).

Outline

CameLIGO

- OCaml-like functional language for the Tezos platform.
- Supports OCaml, JavaScript, Pascal and ReasonML syntax.
- Compiles to Michelson — a stack-based functional language.
- Easy to experiment with contracts using the online-editor.
  https://ide.ligolang.org/

CameLIGO: 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.
- Contracts are functions taking current state as input and producing new state and a list of operations.
- Supports two effectfull operations:
  reading other contracts' state and exceptions.

CameLIGO: program structure

```
(* Storage type *)
type storage = <TYPE>

(* Type of messages *)
type parameter = <TYPE>

(* Return type *)
type return = operation list * storage

<... local declarations ...>

(* Main access point that dispatches to the entrypoints according to the
      smart contract parameter. *)
let main (action, store : parameter * storage) : return =
    <BODY>
```

CameLIGO: a counter contract

```
type storage = int

type parameter =
  Increment of int
| Decrement of int
| Reset

type return = operation list * storage

let add (store, delta : storage * int) : storage = store + delta
let sub (store, delta : storage * int) : storage = store - delta

let main (action, store : parameter * storage) : return =
  let ops = ([] : operation list) in
  match action with
    Increment (n) -> (ops, add (store, n))
  | Decrement (n) -> (ops, sub (store, n))
  | Reset         -> (ops, 0)
```

CameLIGO: data types

- Usual primitive types: `unit`, `bool`, `nat`, `int`, `string` ...
- Blockchain-specific primitive types:
- `tez`, `key`, `signature`, `address`, `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`.

Liquidity: language constructs

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

```
type full_name = string * string
let alice_johnson : full_name = ("Alice", "Johnson")
let (first_name, last_name) : full_name = alice_johnson
let first_name : string = alice_johnson.0
```

- Records

  creation and field access

```
let alice : user = {id = 1n; is_admin = true; name = "Alice"}
let alice_admin : bool = alice.is_admin
```

  field "update" syntax

```
let alince_new_id : user = {alice with id=2n}
```

- General recursion is supported, but tail-recursion only.
- More details here: https://ligolang.org/docs/intro/introduction

CameLIGO: calling other contracts

- Calls are results of the execution of your contract along with the new state.
- Assume that the counter contract is deployed at the address `"KT19wgxcuXG9VH4Af5Tpm1vqEKdaMFpznXT3"`.
- We can send an `Incremet` call as follows.

```
type return = operation list * storage

let main (inc, store : int * storage) : return =
  let counter_addr = ("KT19wgxcuXG9VH4Af5Tpm1vqEKdaMFpznXT3" :
       address) in
  match (Tezos.get_contract_opt counter_addr : parameter contract
       option) with
    Some counter ->
      let op = Tezos.transaction (Increment inc) 0tez counter in
      ([op], store)
  | None -> (failwith "Contract not found." : return)
```

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).
- Polymorphic functions and parametric data types are supported, but they will be monomorphised in Michelson.

Some compiler passes:

- conversion of the partly-imperative surface syntaxes to a common pure IR;
- converting tail recursion to loops;
- eliminating the module system;
- monomorphisation;
- uncurrying;
- optimisations similar to Appel & Jim's Shrinking Lambda Expressions in Linear Time.

DEMO

`https://ide.ligolang.org/`

Outline

Smart contracts: an overview

CameLIGO

Concordium's Rust Smart Contracts

Exercises

Rust

- A hybrid paradigm general-purpose programming language.
- Has a well-defined functional subset:
  - closures and higher-order functions;
  - algebraic data types;
  - polymorphism;
  - type inference
- A language for systems programming: efficiency and precise control of resources (memory).

Smart Contracts in Concordium

- Rust is a primary smart contract language for Concordium.
- The actual code that is executed on-chain is WebAssembly.
- Rust contracts compiles to WebAssembly using the standard Rust compiler.
- `concordium-std` connects Rust code with the Concordium infrastructure.
- Emitted WebAssembly code is subject to some limitations.

Smart Contracts in Concordium: Piggy bank[5]

- anyone can insert money;
- only the owner can "smash" it and get the amount;
- once smashed, inserting is not possible

[5] See Piggy bank tutorial `https://github.com/Concordium/concordium-rust-smart-contracts/tree/main/examples/piggy-bank`

Smart Contracts in Concordium: an example

```
use concordium_std::*;

enum PiggyBankState { Intact, Smashed }

fn piggy_insert<A: HasActions>(_ctx: &impl HasReceiveContext, _amount:
     Amount, state: &mut PiggyBankState) -> ReceiveResult<A>
{  ensure!(*state == PiggyBankState::Intact);
   Ok(A::accept()) }

fn piggy_smash<A: HasActions>(ctx: &impl HasReceiveContext, state: &mut
     PiggyBankState) -> ReceiveResult<A>
{   let owner = ctx.owner();
    let sender = ctx.sender();
    ensure!(sender.matches_account(&owner));
    ensure!(*state == PiggyBankState::Intact);
    *state = PiggyBankState::Smashed;
    let balance = ctx.self_balance();
    Ok(A::simple_transfer(&owner, balance)) }
```

Outline

Exercises

Exercises, homework and some supplementary materials:

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

Suggested plan

- Pick which language you want to try:

Suggested plan

- Pick which language you want to try:
  - CameLIGO
    - online IDE (https://ide.ligolang.org/)
    - possible to test, deploy and interact with contracts.

Suggested plan

- Pick which language you want to try:
    - CameLIGO
        - online IDE (https://ide.ligolang.org/)
        - possible to test, deploy and interact with contracts.
    - Rust
        - if you have some experience with Rust;
        - a bit more to get started: no online IDE, deploying on the testnet is harder.

Suggested plan

- Pick which language you want to try:
    - CameLIGO
        - online IDE (https://ide.ligolang.org/)
        - possible to test, deploy and interact with contracts.
    - Rust
        - if you have some experience with Rust;
        - a bit more to get started: no online IDE, deploying on the testnet is harder.
- Solve warm-up exercises (model solutions are in the repo/archive).
- Solve homework exercises.

Send your questions and feedback

You are welcome to send your questions and solutions to
daan@cs.au.dk

That's it for today!