Towards safer smart contract languages

Danil Annenkov

Aarhus University, Concordium Blockchain Research Center

Datalogforeningen meeting, November 2, 2019

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:

Smart contracts are programs in a general purpose language running “on a blockchain”.

Smart contracts are neither

At least, currently:

Smart contracts are programs in a general purpose language running "on a blockchain".

Why neither smart nor contracts?

- Connection to the legal contracts is 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: The Evolution

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

Ethereum and Solidity

- Solidity is a high level java/javascript-like imperative language.
- One of the most widely used smart contract languages.
- 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 their methods and sending money.**
- **Calls can happen in any point of the program execution (causes reentrancy issues).**

Is Solidity really solid?

Plenty of vulnerabilities have been found:

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

Towards safer smart contract languages

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

- Many smart contract developers 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

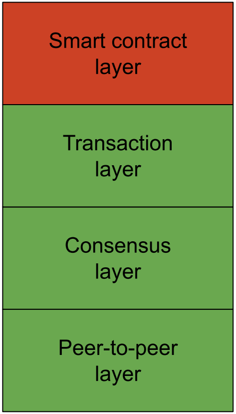
Why designing safe smart contract languages is crucially important?
At least, because:

- Many smart contract developers 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!

The smart contract layer

- Imagine, we have verified all other layers.
- We put some badly designed language on top.
- It's like having your pension depend on a javascript program.
- And any bug is law!



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.

Theoretical foundation of: Haskell, OCaml, Standard ML, Elm, F#, ...

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:
  $\texttt{contract} : \texttt{state} * \texttt{parameters} \rightarrow \texttt{state} * \texttt{operation list}$

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` $\rightarrow$ `state` $*$ `operation list`

Examples of languages with the functional "core".

- Simplicity
- Plutus
- **Liquidity**
- Scilla
- **Oak**

Liquidity demo

DEMO

Oak[1]

- A language for defining smart contracts for the Concordium blockchain.
- The means for developers to interface with the Concordium infrastructure.
- A fork of Elm, a purely functional programming language for web development.
- Prioritises good error messages.
- Static typing.

[1] Thanks Tom Davies for this and the next slide

Oak and Acorn

Oak — convenient, user-friendly	Acorn — internal, efficient
ML^F type system with type inference, user defined data types	Higher-rank polymorphism, explicit typing, no type inference
Intuitive string naming of types and terms	Compact de Bruijn indexing of types and terms
Syntactic sugar (binary operators, if-expressions, etc.)	Concise language with little/no redundancy
Expressive and convenient pattern matching	Predictable performance of pattern matching
Concrete syntax is a key feature	Concrete syntax is a convenience
User friendly tooling and errors	API, virtual machine

Towards formal verification

Proof assistants — special software for developing machine-checkable proofs.

- Allows for developing proofs for mathematics and computer science.
- Proofs are developed by interacting with users.
- Proof automation: tactics, decision procedures, SAT/SMT integration.

Towards formal verification

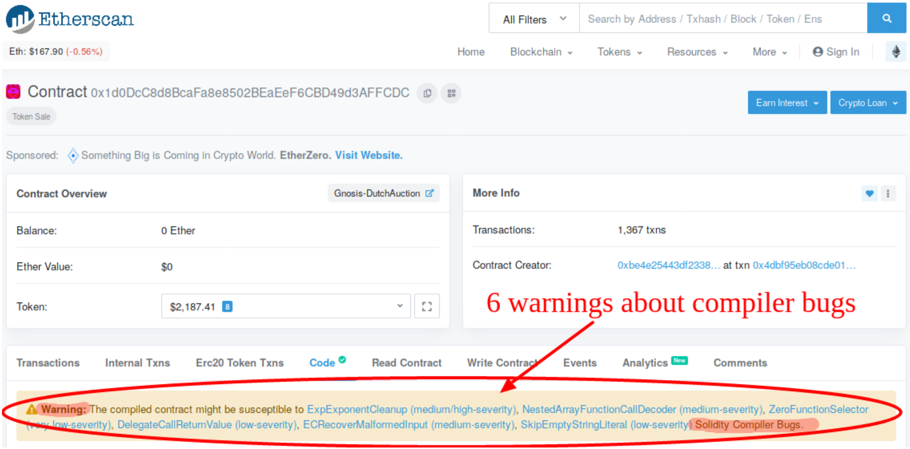
Proof assistants — special software for developing machine-checkable proofs.

- Allows for developing proofs for mathematics and computer science.
- Proofs are developed by interacting with users.
- Proof automation: tactics, decision procedures, SAT/SMT integration.

In particular:

- Formalisation of the programming language's meta-theory.
- Proving correctness of compilers, interpreters, type checking/type inference, etc.
- "Extraction" of bug-free implementation.

Compiler correctness matters



Proof assistants

Proof assistants like Coq, Isabelle/HOL have been successfully applied in large-scale projects

- CompCert — verified C compiler.
- CakeML — verified implemtation of Standard ML.
- seL4 — formal verification of an OS kernel.

Smart contracts formalisation

- Simplicity language — simple language formalised in Coq.
- Ongoing project at *Concordium Research Center*: formalisation of a more expressive smart contract language: the *Oak* language.

ConCert: a smart contract verification framework in Coq

At the Concordium Blockchain Reserach Center, we develop **ConCert** (jww. Bas Spitters and Jakob Botsch Nielsen):

- Verification of *functional* smart contract landuages.
- Particularly, verification of smart contracts in Oak/Acorn.
- Can be used to verify both properties of a smart contract language and properties of concrete smart contracts.
- Allows for verifying properties of interacting smart contracts.

What we can verify?

Crowdfunding: a smart contract allowing arbitrary users to donate money within a deadline.

- Will the users get their money back if the campaign is not funded (goal is not reached)?
- Can the owner withdraw money if goal is reached and deadline have passed?
- Are all contributions recorded correctly in the contract?
- Does contact have enough money at the account to cover all contributions?
- ...

Example: a counter

Acorn

```
data CState = CState [Int64, {address}]

definition owner (s :: CState) =
   case s of
     CState _ d → d

definition balance (s :: CState) =
   case s of
     CState x _ → x

definition count (s :: CState) (msg :: Msg) =
  case msg of
    Inc a →
      CState (Prim.plusInt64 (balance s) a)
             (owner s)
    Dec a →
      CState (Prim.minusInt64 (balance s) a)
             (owner s)
```

Coq

```
Inductive CState :=
  CState_coq : Z → string → CState.

Definition owner : CState →
string := fun x ⇒
  match x with
  | CState_coq _ x1 ⇒ x1
  end.

Definition balance : CState →
Z := fun x ⇒
  match x with
  | CState_coq x0 _ ⇒ x0
  end.

Definition count
  : CState → Msg → CState := fun x x0 ⇒
 match x0 with
 | Inc_coq x1 ⇒
     CState_coq (plusInt64 (balance x) x1)
                (owner x)
 | Dec_coq x1 ⇒
     CState_coq (minusInt64 (balance x) x1)
                (owner x)
 end.
```

Properties of the counter

Sending the "increment" message updates the counter correctly:

```
Lemma inc_correct init_state n i final_state :
  (* precondition *)
  balance init_state = n →

  (* sending "increment" *)
  count init_state (Inc_coq i) = final_state →

  (* result *)
  balance final_state = n + i.
Proof.
  intros Hinit Hrun. subst. reflexivity.
Qed.
```

> Programming languages semanticists should be the obstetricians of programming languages, not their coroners.
>
> — John C. Reynolds