Verifying, testing and running smart contracts in ConCert

Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen and Bas Spitters

Aarhus University, Concordium Blockchain Research Center

The Coq Workshop 2020, July 6

COBRA
CONCORDIUM BLOCKCHAIN
RESEARCH CENTER AARHUS

AARHUS
UNIVERSITY

What are smart contracts?

Programs in a general-purpose language running “on a blockchain”

Blockchain $\sim$ database, smart contracts $\sim$ stored procedures.

What are smart contracts?

Programs in a general-purpose language running "on a blockchain"

Blockchain $\sim$ database, smart contracts $\sim$ stored procedures.

What is so special about smart contracts?

- They often manage money: auctions, crowdfunding campaigns, multi-signature wallets, DAOs.
- Once deployed, contract code cannot be changed.
- Code is Law.
- Can call other contracts containing possibly malicious code.
- Flaws may result in huge financial losses:
    - The DAO $\sim$ \$50M — hacker attack.
    - Parity's multi-signature wallet $\sim$ \$280M — a bug in the library code.

Functional smart contract languages

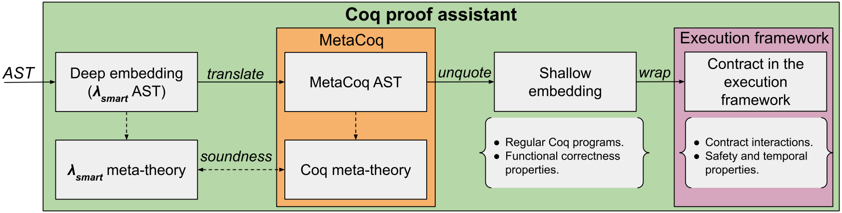
- Contracts are programs in a functional language transforming the state:

  `contract : CallCtx * Msg * State -> State * Action list`
- But blockchains are stateful.
- Contracts are used as transition functions.
- A *scheduler* handles transfers and calls to other contracts in `Action list`.

Examples of such languages:

- LIGO (Tezos)
- **Liquidity** (Dune)
- Scilla (Zilliqa)
- **Midlang**/Retlang (Concordium)
- ...

ConCert: A Smart Contract Certification Framework



DA, Jakob Botsch Nielsen, Bas Spitters. ConCert: A Smart Contract Certification Framework, CPP'20.

Jakob Botsch Nielsen and Bas Spitters. Smart Contract Interactions in Coq. FMBC'19.

- Embedding of a functional smart contract language in Coq.
- Soundness of the embedding through MetaCoq.
- Execution model.
- Verification of a crowdfunding, congress, importing Retlang code.

Our contributions

We extend ConCert:

- Implement extraction to functional smart contract languages.
  - Liquidity (Dune).
  - Midlang (Concordium) (bonus — Elm!).
- Verify complex smart contracts: boardroom voting.
- Add property-based testing (using QuickChick).

Extraction

- Coq supports extraction to OCaml, Haskell and Scheme.
- General idea: turn all parts of a program that do not contribute to computation into $\square$ (a **box**).
- The underlying theory: Pierre Letouzey's thesis.
- Not directly suitable for functional smart contract languages: syntactic and semantic differences.
- Current Coq extraction is not verified.
- MetaCoq erasure is verified![1]

[1]Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau and Théo Winterhalter. Coq Coq correct! verification of type checking and erasure for Coq, in Coq

Smart contract extraction: challenges

- No `Obj.magic`/`unsafeCoerce`
- Non-recursive data types only.
- Limited support for recursion (e.g. tail recursion only, or no direct access to recursion — only through primitives).

Smart contract extraction: challenges

- No `Obj.magic`/`unsafeCoerce`
- Non-recursive data types only.
- Limited support for recursion (e.g. tail recursion only, or no direct access to recursion — only through primitives).

Consequences:

- Some extracted code will not be well-typed.
- Remapping (cf. `Extract Constant`) is mandatory for some recursive definitions.

MetaCoq verified erasure

- A translation from CIC (Calculus of Inductive Constructions) into $CIC_{\square}$.
- Provides a proof that the evaluation of the original and the erased terms agree.

Missing bits for the practical use:

- No erasure for types and inductives.
- No optimisations (e.g. removing boxes).

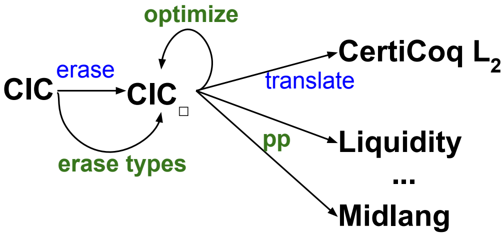
MetaCoq verified erasure

- A translation from CIC (Calculus of Inductive Constructions) into $\mathrm{CIC}_{\square}$.
- Provides a proof that the evaluation of the original and the erased terms agree.

Missing bits for the practical use:

- No erasure for types and inductives.
- No optimisations (e.g. removing boxes).

We implement the missing bits an develop pretty-printers directly in Coq.



MetaCoq verified erasure: deboxing

```
Definition square (xs : list nat) : list nat :=
        @map nat nat (fun x : nat ⇒ x ∗ x) xs.
```

Erases to (implicit type arguments become boxes):

```
fun xs ⇒ Coq.Lists.List.map □□ (fun x ⇒ Coq.Init.Nat.mul x x) xs
```

We want to remove the boxes:

```
fun xs ⇒ Coq.Lists.List.map (fun x ⇒ Coq.Init.Nat.mul x x) xs
```

This removes redundant computations and makes remapping `Coq.Lists.List.map` to a target language `map` easier.

MetaCoq verified erasure: deboxing

```
Definition square (xs : list nat) : list nat :=
      @map nat nat (fun x : nat ⇒ x ∗ x) xs.
```

Erases to (implicit type arguments become boxes):

```
fun xs ⇒ Coq.Lists.List.map □□(fun x ⇒ Coq.Init.Nat.mul x x) xs
```

We want to remove the boxes:

```
fun xs ⇒ Coq.Lists.List.map (fun x ⇒ Coq.Init.Nat.mul x x) xs
```

This removes redundant computations and makes remapping `Coq.Lists.List.map` to a target language `map` easier.

Deboxing — a transformation that removes redundant boxes.

Deboxing as an optimisation

When (and why) is it safe to remove boxes?

- Boils down to: $(\texttt{fun x} \Rightarrow \texttt{t})\ \texttt{v} \sim \texttt{t}$, if `x` does not occur free in `t`
- Deboxing is a special case: $(\texttt{fun A x} \Rightarrow \texttt{t})\ \square \sim (\texttt{fun x} \Rightarrow \texttt{t})$. From erasure, we know that `A` does not occur free in `t`.
- We remove boxes from applications of constants (e.g. `map`) and constructors.
- Boxes coming from any "logical" parts (types or propositions) can be removed in the same way.

Deboxing as an optimisation

When (and why) is it safe to remove boxes?

- Boils down to: `(fun x ⇒ t) v` $\sim$ `t`, if `x` does not occur free in `t`
- Deboxing is a special case: `(fun A x ⇒ t) □` $\sim$ `(fun x ⇒ t)`. From erasure, we know that `A` does not occur free in `t`.
- We remove boxes from applications of constants (e.g. `map`) and constructors.
- Boxes coming from any "logical" parts (types or propositions) can be removed in the same way.

Caveats

- Partial applications might require eta-expansion.
- With eta-expansion, it might not be an optimisation.

Deboxing as an optimisation

When (and why) is it safe to remove boxes?

- Boils down to: `(fun x ⇒ t) v` $\sim$ `t`, if `x` does not occur free in `t`
- Deboxing is a special case: `(fun A x ⇒ t) □` $\sim$ `(fun x ⇒ t)`. From erasure, we know that `A` does not occur free in `t`.
- We remove boxes from applications of constants (e.g. `map`) and constructors.
- Boxes coming from any "logical" parts (types or propositions) can be removed in the same way.

Caveats

- Partial applications might require eta-expansion.
- With eta-expansion, it might not be an optimisation.

We implement the general case: eta-expand, remove arguments and abstractions that do no occur.

A counter contract

```
Definition storage := Z.
Definition pos := {z : Z | 0 <? z}.
Inductive msg :=
  Inc (_ : Z)
| Dec (_ : Z).

Program Definition inc_counter (st : storage) (inc : pos) :
  {new_st : storage | st <? new_st} :=
  st + proj1_sig inc. Next Obligation. (* proof omitted *) Qed.
...

Definition counter (msg : msg) (st : storage)
  : option (list SimpleActionBody * storage) :=
  match msg with
  | Inc i => match (bool_dec (0 <? i) true) with
             | left h => Some ([], proj1_sig (inc_counter st (exist i h)))
             | right _ => None
             end
  | Dec i => ...
  end.
```

Extracted code

```
type storage = int
type coq_msg =
    Coq_Inc of int
  | Coq_Dec of int



let exist a = a



let coq_inc_counter (st : storage) (inc : int)
 = exist (addInt st ((fun x → x) inc))
...


let coq_counter (msg : coq_msg) (st : storage)
 = match msg with
  Coq_Inc i →
    (match coq_bool_dec (ltInt 0 i) true with
       Coq_left  →
         Some ([],  (fun x → x)
              (coq_inc_counter st (exist i)))
      | Coq_right  → None)
| Coq_Dec i → ...
```

Listing 1: Liquidity

```
type alias Storage = Int
type Msg
  = Inc Int
  | Dec Int

type alias Pos = Sig Int

type Sig a = Exist a
proj1_sig : Sig a → a
proj1_sig e = case e of Exist a → a

inc_counter : Storage → Pos → Sig Storage
inc_counter st inc = Exist (add st (proj1_sig inc))
...
counter : Msg → Storage
               → Option (Prod Transaction Storage)
counter msg st =
  case msg of
    Inc i →
      case bool_dec (lt 0 i) True of
        Left → Some (Pair Transaction.none
          (proj1_sig (inc_counter st (Exist i))))
        Right →
          None
    Dec i → ...
```

Listing 2: Midlang

Extracted code

```
type storage = int
type coq_msg =
    Coq_Inc of int
  | Coq_Dec of int



let exist a = a



let coq_inc_counter (st : storage) (inc : int)
 = exist (addInt st ((fun x → x) inc))
...


let coq_counter (msg : coq_msg) (st : storage)
 = match msg with
  Coq_Inc i →
   (match coq_bool_dec (ltInt 0 i) true with
      Coq_left →
        Some ([], (fun x → x)
             (coq_inc_counter st (exist i)))
    | Coq_right → None)
| Coq_Dec i → ...
```

Listing 1: Liquidity

```
type alias Storage = Int
type Msg
  = Inc Int
  | Dec Int

type alias Pos = Sig Int

type Sig a = Exist a
proj1_sig : Sig a → a
proj1_sig e = case e of Exist a → a

inc_counter : Storage → Pos → Sig Storage
inc_counter st inc = Exist (add st (proj1_sig inc))
...
counter : Msg → Storage
          → Option (Prod Transaction Storage)
counter msg st =
  case msg of
    Inc i →
      case bool_dec (lt 0 i) True of
        Left → Some (Pair Transaction.none
          (proj1_sig (inc_counter st (Exist i))))
        Right →
          None
    Dec i → ...
```

Listing 2: Midlang

Extracted code

```
type storage = int
type coq_msg =
    Coq_Inc of int
  | Coq_Dec of int
```

Ad-hoc remapping for sig

```
let exist a = a



let coq_inc_counter (st : storage) (inc : int)
 = exist (addInt st ((fun x → x) inc))
...


let coq_counter (msg : coq_msg) (st : storage)
 = match msg with
  Coq_Inc i →
    (match coq_bool_dec (ltInt 0 i) true with
       Coq_left  →
         Some ([],  (fun x → x)
            (coq_inc_counter st (exist i)))
      | Coq_right  → None)
| Coq_Dec i → ...
```

Listing 1: Liquidity

```
type alias Storage = Int
type Msg
  = Inc Int
  | Dec Int

type alias Pos = Sig Int

type Sig a = Exist a
proj1_sig : Sig a → a
proj1_sig e = case e of Exist a → a

inc_counter : Storage → Pos → Sig Storage
inc_counter st inc = Exist (add st (proj1_sig inc))
...
counter : Msg → Storage
          → Option (Prod Transaction Storage)
counter msg st =
  case msg of
    Inc i →
      case bool_dec (lt 0 i) True of
        Left → Some (Pair Transaction.none
          (proj1_sig (inc_counter st (Exist i))))
        Right →
          None
    Dec i → ...
```

Listing 2: Midlang

Experience with extraction

- We have extracted several smart contracts: counter, crowdfunding, prototype DSL interpreter, escrow.
- Liquidity has many restrictions: uncurried constructors, tail recursion only, recursive functions take a single argument, no pattern-matching on tuples ...
- Midlang is closer to $\mathrm{CIC}_{\square}$, extraction is more principled.
- Current TCB: MetaCoq spec + MetaCoq quote + pretty-printing.
- Soundness proofs for optimisations are in progress.
- Ideally: have semantics of both languages formalised.

Boardroom voting

- Contract for small-scale anonymous e-voting, based on Open Vote Network[2]
- Parties prove they are following the protocol in zero-knowledge
- At the end, the contract can compute a public tally from private votes
- We prove that if parties follow the protocol, computed tally is correct

```
Inductive Msg :=
| signup (pk : A) (proof : A * Z)
| commit_to_vote (hash : positive)
| submit_vote (v : A) (proof : VoteProof)
| tally_votes.
```

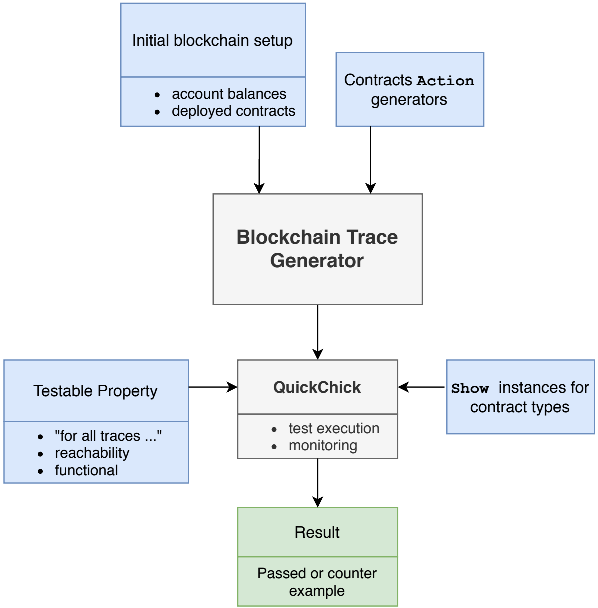
Listing 3: Boardroom voting messages

[2] Feng Hao, Peter Y. A. Ryan, and Piotr Zieliński. "Anonymous voting by two-round public discussion".

Boardroom voting

- Following the protocol means: their messages and proofs are created with our functions, they are timely (all parties have registered once the registration period is over).
- Formalization required some finite field arithmetic
  - Would have liked to use Mathcomp, but felt more like a framework than a library (e.g. own definition of gcd, no proof about correspondence to `Nat.gcd`)
  - Thus we rolled the parts we needed ourselves
- Future: integration with optimized cryptographic implementations using fiat-crypto.
- Future: move to elliptic curves (more efficient), extraction

Overview of the Testing Framework



Testing Smart Contracts

- Our approach is based on generating blockchain execution traces.
- Allows for stating functional correctness, safety and temporal properties.
- Case studies: ERC20 Token, FA2 Token, Congress, UniSwap.
- We have (re-)discovered many known vulnerabilities/bugs.
- Allows for finding bugs earlier, helping the verification efforts.

Testing Smart Contracts

- Our approach is based on generating blockchain execution traces.
- Allows for stating functional correctness, safety and temporal properties.
- Case studies: ERC20 Token, FA2 Token, Congress, UniSwap.
- We have (re-)discovered many known vulnerabilities/bugs.
- Allows for finding bugs earlier, helping the verification efforts.

Tricky bits

- Requires specialised generators defined manually (otherwise too many discards).

Conclusions

- Extraction to Liquidity and Midlang/Elm using MetaCoq's certified erasure.
- Verification of a boardroom voting contract that uses crypto functionality.
- Integration of QuickChick for smart contract testing.
- Our development: `https://github.com/AU-COBRA/ConCert`

Conclusions

- Extraction to Liquidity and Midlang/Elm using MetaCoq's certified erasure.
- Verification of a boardroom voting contract that uses crypto functionality.
- Integration of QuickChick for smart contract testing.
- Our development: `https://github.com/AU-COBRA/ConCert`

What we would like to improve in the Coq infrastructure

- Fully certified extraction.
- Extraction **framework** with some intermediate language ($CIC_{\square}$?, mini-ml?) — cf. A Code Generator Framework for Isabelle/HOL.
- More features become "standard" (stdlib2?).