COBRA
CONCORDIUM BLOCKCHAIN
RESEARCH CENTER AARHUS

AARHUS
UNIVERSITY

Smart contracts in a proof assistant

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

The COBRA Seminar, October 5, 2021.

Aarhus University, Concordium Blockchain Research Center

Types

- What are types for?

Types

- What are types for?
- To write specs!

```
int compute (int n, int m) {
  n + n/m;
}
```

- What we can say about `compute` by its signature?

Types

- What are types for?
- To write specs!

```
int compute (int n, int m) {
  n + n/m;
}
```

- What we can say about `compute` by its signature?
- Not much:
    - takes two (32-bit signed) **integer numbers**;
    - returns an **integer number**.

Types

- What are types for?
- To write specs!

```
int compute (int n, int m) {
  n + n/m;
}
```

- What we can say about `compute` by its signature?
- Not much:
  - takes two (32-bit signed) **integer numbers**;
  - returns an **integer number**.
- It is helpful: `compute("blah",true)` is rejected by the compiler.

Types

- What are types for?
- To write specs!

```
int compute (int n, int m) {
  n + n/m;
}
```

- What we can say about `compute` by its signature?
- Not much:
  - takes two (32-bit signed) **integer numbers**;
  - returns an **integer number**.
- It is helpful: `compute("blah",true)` is rejected by the compiler.
- But not so much, sometimes:
  `compute(42,0)` is well-typed, but fails at run-time.

Types

- What are types for?
- To write specs!

```
int compute (int n, int m) {
  n + n/m;
}
```

- What we can say about `compute` by its signature?
- Not much:
  - takes two (32-bit signed) **integer numbers**;
  - returns an **integer number**.
- It is helpful: `compute("blah",true)` is rejected by the compiler.
- But not so much, sometimes:
  `compute(42,0)` is well-typed, but fails at run-time.
- Can we do better?

Proof assistants

Proof assistants — software for developing machine-checkable proofs.

- For mathematics and computer science.
- Proofs are developed by interacting with users.
- Proof automation: tactics, decision procedures, SMT integration.

Proof assistants

Proof assistants — software for developing machine-checkable proofs.

- For mathematics and computer science.
- Proofs are developed by interacting with users.
- Proof automation: tactics, decision procedures, SMT integration.

Some application in CS:

- Proving correctness of compilers, type checking/type inference, etc.
- **Program verification**.
- **"Extraction" of the bug-free implementation.**

The Coq proof assistant

- Coq means “rooster” in French.
- Mature system: more than 30 year!
- Used in many project in CS and mathematics:
  CompCert, Four color theorem, Feit-Thompson theorem, ...
- Based on dependent types: can express specs in types!
- Widely used in COBRA verification projects.

Notation

Java-like syntax

```
int my_func (int n, int m) {
  ...
}
```

Coq syntax

```
Definition my_func (n m : Z) : Z :=
  ...
```

- we write `Z` for the integer type;
- `n : Z` means that `n` is an integer.

Subset types in Coq

- Coq's version of refinement types.
- Types with a predicate.
- Notation: `{x : A | P x}`.
- Elements of `A` that satisfy the predicate `P`.

Subset types in Coq

- Coq's version of refinement types.
- Types with a predicate.
- Notation: `{x : A | P x}`.
- Elements of `A` that satisfy the predicate `P`.
- Elements are essentially value-proof pairs with projections
  - `proj1_sig` — returns a value of type `A`;
  - `proj2_sig` — returns a proof that `P` hold for the value.

Subset types in Coq

- Coq's version of refinement types.
- Types with a predicate.
- Notation: {`x : A | P x`}.
- Elements of `A` that satisfy the predicate `P`.
- Elements are essentially value-proof pairs with projections
  - `proj1_sig` — returns a value of type `A`;
  - `proj2_sig` — returns a proof that `P` hold for the value.
- Examples:
  - {`n : nat | 0 < n`} — positive natural numbers;
  - {`i : Z | 0 <> i`} — non-zero integers;
  - {`xs : list A | 0 < length xs`} — non-empty lists.

Subset types in Coq

- Coq's version of refinement types.
- Types with a predicate.
- Notation: `{x : A | P x}`.
- Elements of `A` that satisfy the predicate `P`.
- Elements are essentially value-proof pairs with projections
  - `proj1_sig` — returns a value of type `A`;
  - `proj2_sig` — returns a proof that `P` hold for the value.
- Examples:
  - `{n : nat | 0 < n}` — positive natural numbers;
  - `{i : Z | 0 <> i}` — non-zero integers;
  - `{xs : list A | 0 < length xs}` — non-empty lists.
- `Program` environment for managing proof obligations.
- Convenient for the correct-by-construction approach.

We can do better now

We specify the invariant for division in the type

```
Program Definition safe_div (n : Z) (m : { i : Z | i <> 0}) : Z := ...
Definition safe_compute (n : Z) (m : { i : Z | i <> 0}) : Z :=
  n + safe_div n m.
```

We can do better now

We specify the invariant for division in the type

```
Program Definition safe_div (n : Z) (m : { i : Z | i <> 0}) : Z := ...
Definition safe_compute (n : Z) (m : { i : Z | i <> 0}) : Z :=
  n + safe_div n m.
```

For each call of `safe_compute`, we must **provide a proof of the invariant**.

```
Program Definition compute_power_2 (n m : nat) : Z :=
  safe_compute (2^n) (2^m).
(* Here we prove that 2^m <> 0 *)
```

We can do better now

We specify the invariant for division in the type

```
Program Definition safe_div (n : Z) (m : { i : Z | i <> 0}) : Z := ...
Definition safe_compute (n : Z) (m : { i : Z | i <> 0}) : Z :=
  n + safe_div n m.
```

For each call of `safe_compute`, we must **provide a proof of the invariant**.

```
Program Definition compute_power_2 (n m : nat) : Z :=
  safe_compute (2^n) (2^m).
(* Here we prove that 2^m <> 0 *)
```

A validation barrier at the entry point:
validate user input once and for all, the rest works on valid data

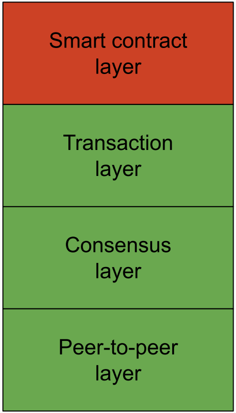
```
Program Definition public_compute (n m : Z) : option Z :=
  match is_zero m with
  | right p ⇒ Some (safe_compute n m)
  | left p ⇒ None (* signals that the validation has failed *)
  end.
```

NOTE: `option Z` means it is a partial function.

Smart contracts

Programs deployed on a blockchain

- Transaction protocols between parties.
- Use the underlying blockchain infrastructure.
- Often targeted by hackers.



Functional smart contract languages

- Contracts are (partial) state transition functions:

  $\texttt{contract} : \texttt{CallCtx} * \texttt{Msg} * \texttt{State} \to \texttt{option}\ (\texttt{State} * \texttt{list Action})$

  - takes a triple (`call_ctx`, `msg`, `st`);
  - either returns a tuple (`st, actions`), or fails.

Functional smart contract languages

- Contracts are (partial) state transition functions:

  $\texttt{contract} : \texttt{CallCtx} * \texttt{Msg} * \texttt{State} \rightarrow \texttt{option}\ (\texttt{State} * \texttt{list Action})$

  - takes a triple (`call_ctx`, `msg`, `st`);
  - either returns a tuple (`st`, `actions`), or fails.
- A *scheduler*
  - updates the state;
  - handles transfers and calls to other contracts in `Action list`.

Fits well with modern blockchains: Concordium, Tezos, Dune.

ConCert: A Smart Contract Certification Framework

- Infrastructure for developing smart contracts in Coq.
- The execution layer (formalises the scheduler).
- Smart contract verification infrastructure.
- Generation of executable code for several target platforms: Concordium, Tezos, Dune.

ConCert: A Smart Contract Certification Framework

- Infrastructure for developing smart contracts in Coq.
- The execution layer (formalises the scheduler).
- Smart contract verification infrastructure.
- Generation of executable code for several target platforms: Concordium, Tezos, Dune.
- Verified: token standards implementations, escrow, crowdfunding ...
- Most recent (by Eske Hoy Nielsen): Dexter — a decentralised exchange for Tezos.
- WIP: formalisation of the Concordium's token standard (CTS).

The counter contract

```
Definition State := Z.
(* State inc_counter (State prev_st, int inc) { return (prev_st + inc) } *)
Program Definition inc_counter (prev_st : State) (inc : Z) : State
    := prev_st + inc.
```

```
Program Definition dec_counter (...) := ...
```

The counter contract

```
Definition State := Z.

Program Definition inc_counter
    (prev_st : State) (* take the previous state *)
    (inc : Z) :       (* increment by [inc] *)
    State             (* return a new (incremented!) state *)
    := prev_st + inc.


Program Definition dec_counter (...) := ...
```

The counter contract

```
Definition State := Z.

Program Definition inc_counter
    (prev_st : State)
    (inc : {z : Z | 0 < z }) :
    { new_st : State | prev_st < new_st ∧ new_st = prev_st + inc }
    := prev_st + inc.
    (* the proof is constructed using proof automation *)

Program Definition dec_counter (...) := ...
```

The counter contract

```
Definition State := Z.

Program Definition inc_counter
    (prev_st : State)
    (inc : {z : Z | 0 < z}) :
    { new_st : State | prev_st < new_st ∧ new_st = prev_st + inc}
    := prev_st + inc.
    (* the proof is constructed using proof automation *)

Program Definition dec_counter (...) := ...
```

pre-condition

post-condition

The counter contract

```
Definition State := Z.

Program Definition inc_counter
    (prev_st : State)
    (inc : {z : Z | 0 < z }) :
    { new_st : State | prev_st < new_st ∧ new_st = prev_st + inc }
    := prev_st + inc.
    (* the proof is constructed using proof automation *)

Program Definition dec_counter (...) := ...
```

validation barrier

```
Program Definition counter_receive (msg : Msg) (st : State)
    : option (State * list Action) :=
    match msg with
    | Inc i ⇒ match is_gt_zero i with
              | left H ⇒ Some (inc_counter st i, [])
              | right _ ⇒ None
              end
    | Dec i ⇒ ...
  end.
```

More properties!

- Subset types are convenient for invariants on the contract's state.

More properties!

- Subset types are convenient for invariants on the contract's state.
- **Crowdfunding:**
  individual contributions and the total balance agree.

```
Record State := { contribs : Map Address Amount;
                  total : Amount;
                  valid : sum contribs = total }.
```

More properties!

- Subset types are convenient for invariants on the contract's state.
- **Crowdfunding:**
  individual contributions and the total balance agree.

```
Record State := { contribs : Map Address Amount;
                  total : Amount;
                  valid : sum contribs = total }.
```

- **Tokens:**
  balances are preserved.

```
Definition transfer (old_balances : FMap Address Amount)
                    (from : Address) (to : Address) :
                    { new_balances | sum old_balances = sum new_balances}.
```

More theorems!

- Sometimes subset types are not enough:
  - how different functions relate to each other

    `FMap.find k (FMap.add k v m) = Some v`
  - smart contract interactions
  - ...

More theorems!

- Sometimes subset types are not enough:
  - how different functions relate to each other
    `FMap.find k (FMap.add k v m) = Some v`
  - smart contract interactions
  - ...
- Use a different verification style:
  state and proof theorems *after* writing all definitions.

More theorems!

- Sometimes subset types are not enough:
  - how different functions relate to each other
    `FMap.find k (FMap.add k v m) = Some v`
  - smart contract interactions
  - ...
- Use a different verification style:
  state and proof theorems *after* writing all definitions.
- A property for `Counter` on execution traces:

```
Theorem counter_correct : forall init_val state contract_calls,
  reachable state → (* other conditions *)
  state = sum_inc_dec contract_calls init_val.
```

  the contract's state is exactly the sum of all increments and decrements sent to the contract, plus the initial counter value.

Code extraction in Coq

- Now we have a verified smart contract in Coq.
- How we can obtain “actual” SC code from a formal development?

Code extraction in Coq

- Now we have a verified smart contract in Coq.
- How we can obtain "actual" SC code from a formal development?
- Coq features **code extraction: producing executable code in a conventional programming language.**
- Support supports OCaml, Haskell and Scheme out of the box.

Code extraction in Coq

- Now we have a verified smart contract in Coq.
- How we can obtain “actual” SC code from a formal development?
- Coq features **code extraction: producing executable code in a conventional programming language.**
- Support supports OCaml, Haskell and Scheme out of the box.

However:

✗ Does not support smart contract languages.

✗ Current Coq extraction is not verified.

Code extraction in Coq

- Now we have a verified smart contract in Coq.
- How we can obtain “actual” SC code from a formal development?
- Coq features **code extraction: producing executable code in a conventional programming language.**
- Support supports OCaml, Haskell and Scheme out of the box.

However:

✗ Does not support smart contract languages.

✗ Current Coq extraction is not verified.

We address these points in ConCert.

Extraction in ConCert

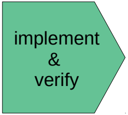
An **extensible** extraction pipeline with **small TCB**

Extraction in ConCert

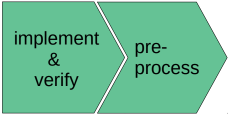
An **extensible** extraction pipeline with **small TCB**

- Implement the extraction pipeline in Coq.
- Use MetaCoq's verified erasure as a basis.
- Add verified pre- and post-processing steps.
- Let the users add transformations/target languages.

The extraction pipeline

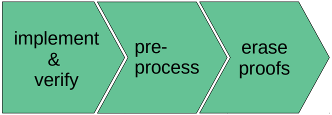


The extraction pipeline



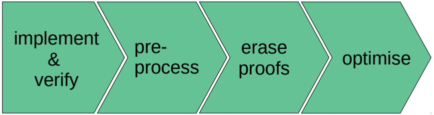
- preprocess: inlining, specialisation ... + generate proofs

The extraction pipeline



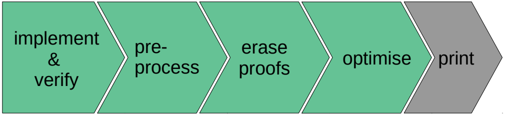
- preprocess: inlining, specialisation ... + generate proofs
- erase proofs: use the extended verified MetaCoq erasure

The extraction pipeline



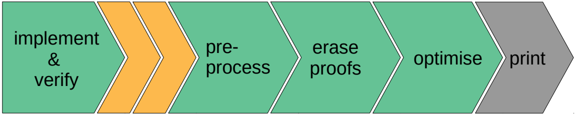
- preprocess: inlining, specialisation ... + generate proofs
- erase proofs: use the extended verified MetaCoq erasure
- optimise: verified dead argument elimination

The extraction pipeline



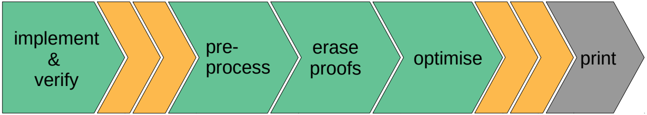
- preprocess: inlining, specialisation ... + generate proofs
- erase proofs: use the extended verified MetaCoq erasure
- optimise: verified dead argument elimination
- print: Rust (Concordium), CameLIGO (Tezos), Liquidity (Dune)

The extraction pipeline



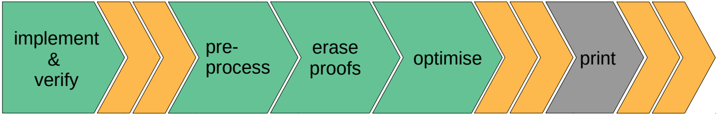
- preprocess: inlining, specialisation ... + generate proofs
- erase proofs: use the extended verified MetaCoq erasure
- optimise: verified dead argument elimination
- print: Rust (Concordium), CameLIGO (Tezos), Liquidity (Dune)
- add new pre-processing steps

The extraction pipeline



- preprocess: inlining, specialisation ... + generate proofs
- erase proofs: use the extended verified MetaCoq erasure
- optimise: verified dead argument elimination
- print: Rust (Concordium), CameLIGO (Tezos), Liquidity (Dune)
- add new pre-processing steps
- add new optmisations

The extraction pipeline



- preprocess: inlining, specialisation ... + generate proofs
- erase proofs: use the extended verified MetaCoq erasure
- optimise: verified dead argument elimination
- print: Rust (Concordium), CameLIGO (Tezos), Liquidity (Dune)
- add new pre-processing steps
- add new optmisations
- add new target languages

Extracted code

```
Inductive sig (A : Type)
           (P : A → Prop) : Type :=
 exist : forall (x : A), P x → {x : A | P x}

 Definition inc_counter
           (prev_st : State)
           (inc : {z : Z | 0 < z }) :
           { new_st : State | ... }

   := prev_st + inc.
```

Extracted code

```
Inductive sig (A : Type)
            (P : A → Prop) : Type :=
 exist : forall (x : A), P x → sig P

Definition inc_counter
            (prev_st : State)
            (inc : sig (fun z ⇒ 0 < z)) :
            sig (fun new_st ⇒ ...)

    := exist
        (Z.add prev_st (proj1_sig inc))
        BIG_PROOF_TERM
```

Extracted code

```
Inductive sig (A : Type)
             (P : A → Prop) : Type :=
 exist : forall (x : A), P x → sig P

Definition inc_counter
             (prev_st : State)
             (inc : sig (fun z ⇒ 0 < z)) :
             sig (fun new_st ⇒ ...)

    := exist
        (Z.add prev_st (proj1_sig inc))
        BIG_PROOF_TERM
```

```
pub enum Sig<A> {
  exist(A)
}

fn inc_counter(&'a self,
               prev_st: State<'a>,
               inc: &'a Sig<i64>)
   → &'a Sig<State<'a>> {
self.alloc(
 Sig::exist(
  self.add(prev_st, self.proj1_sig(inc))))
}
```

Conclusion and future work

- Smart contracts are crucial to get right.
- Expressive type systems in proof assistants: specs in types.
- Use the Coq proof assistant to develop smart contracts.
- Get smart contract code using code extraction.

Conclusion and future work

- Smart contracts are crucial to get right.
- Expressive type systems in proof assistants: specs in types.
- Use the Coq proof assistant to develop smart contracts.
- Get smart contract code using code extraction.
- Expressive type come at a price: proofs.
- Good news: proof automation helps.
- Future work: better proof automation :)

Conclusion and future work

- Smart contracts are crucial to get right.
- Expressive type systems in proof assistants: specs in types.
- Use the Coq proof assistant to develop smart contracts.
- Get smart contract code using code extraction.
- Expressive type come at a price: proofs.
- Good news: proof automation helps.
- Future work: better proof automation :)

ConCert + extraction =

dependent types in your favorite SC language

Thank you for your attention!

Our development on GitHub:

https://github.com/AU-COBRA/ConCert