Extending MetaCoq Erasure: Extraction to Rust and Elm

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

Aarhus University, Concordium Blockchain Research Center

The Coq Workshop, July 2, 2021

Extraction in Coq

- Coq supports extraction to OCaml, Haskell and Scheme.
- General idea: turn computationally irrelevant bits into $\Box$ (a **box**).
- Proofs (propositions) and types appearing in terms become boxes.
- The underlying theory: Pierre Letouzey's PhD thesis.[1]

[1] Certified functional programming : Program extraction within Coq proof assistant.

Extraction in Coq

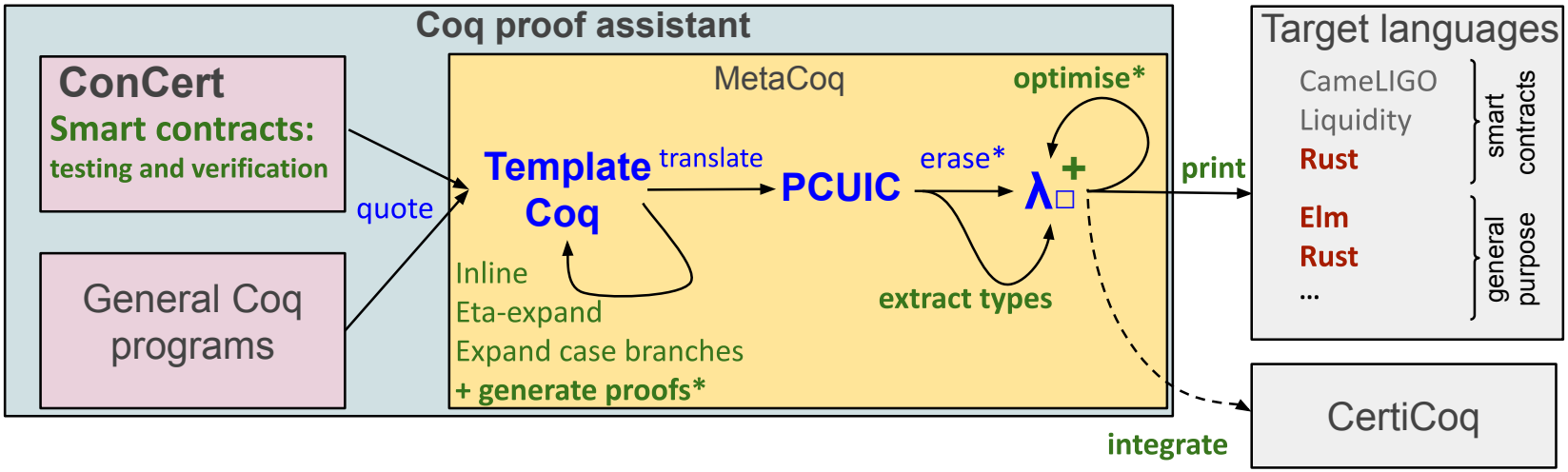
- Coq supports extraction to OCaml, Haskell and Scheme.
- General idea: turn computationally irrelevant bits into □ (a **box**).
- Proofs (propositions) and types appearing in terms become boxes.
- The underlying theory: Pierre Letouzey's PhD thesis.[1]

✗ Does not support target languages we are interested in.
✗ Current Coq extraction is not verified.
✓ MetaCoq erasure is verified![2]

[1] Certified functional programming : Program extraction within Coq proof assistant.

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

Our extraction pipeline



- **green** items — our contributions (including the CPP'21 paper[3]);
- marked with $*$ — verified;
- improvements (this work):
  - pre-processing step with proof-generating transformations;
  - extensible — add your conversion-preserving transformation;
  - new targets — Rust;
  - new use case — verified Elm web app.

[3] DA, Mikkel Milo, Jakob Botsch Nielsen and Bas Spitters.
Extracting Smart Contracts Tested and Verified in Coq

Extraction to Rust

- Rust is getting popular:
  Rust foundation (AWS, google, mozilla, MS, FB, Huawei),
  also: Concordium.

Extraction to Rust

- Rust is getting popular:
  Rust foundation (AWS, google, mozilla, MS, FB, Huawei),
  also: Concordium.
- Multi-paradigm language with manual memory management.

Extraction to Rust

- Rust is getting popular:
  Rust foundation (AWS, google, mozilla, MS, FB, Huawei),
  also: Concordium.
- Multi-paradigm language with manual memory management.
- Has a nice functional fragment:
  - sum/product types;
  - pattern matching;
  - everything is an expression;
  - immutability by default;
  - (almost) Hindley-Milner (without let-polymorphism)

Extraction to Rust: challenges

- Recursive data types: type definitions contain their fields by value.

  ```
  pub enum List<A> { nil, cons(A, List<A>) }
  ```

  has infinite size — invalid in Rust.

  Solution: use references:

  ```
  pub enum List<'a, A> {
    nil(PhantomData<&'a A>),
    cons(PhantomData<&'a A>, A, &'a List<'a, A>)
  }
  ```

Extraction to Rust: challenges

- Recursive data types: type definitions contain their fields by value.

  ```
  pub enum List<A> { nil, cons(A, List<A>) }
  ```

  has infinite size — invalid in Rust.

  Solution: use references:

  ```
  pub enum List<'a, A> {
    nil(PhantomData<&'a A>),
    cons(PhantomData<&'a A>, A, &'a List<'a, A>)
  }
  ```

- No GC. Currently, we use arena/region-based allocation.
  Possible to use existing GC, e.g. Boehm–Demers–Weiser.

Extraction to Rust: challenges

- Recursive data types: type definitions contain their fields by value.

  ```
  pub enum List<A> { nil, cons(A, List<A>) }
  ```

  has infinite size — invalid in Rust.

  Solution: use references:

  ```
  pub enum List<'a, A> {
    nil(PhantomData<&'a A>),
    cons(PhantomData<&'a A>, A, &'a List<'a, A>)
  }
  ```

- No GC. Currently, we use arena/region-based allocation.
  Possible to use existing GC, e.g. Boehm–Demers–Weiser.
- Closures. Supported, but each closure gets a unique type.
  `let c1 = |y| y + 1; let c2 = |z| z + 1; let foo = [c1, c2];` doesn't work.
  Solution: allocate closures, use references.

Extraction to Rust: challenges

- Recursive data types: type definitions contain their fields by value.

  ```
  pub enum List<A> { nil, cons(A, List<A>) }
  ```

  has infinite size — invalid in Rust.

  Solution: use references:

  ```
  pub enum List<'a, A> {
    nil(PhantomData<&'a A>),
    cons(PhantomData<&'a A>, A, &'a List<'a, A>)
  }
  ```

- No GC. Currently, we use arena/region-based allocation.
  Possible to use existing GC, e.g. Boehm–Demers–Weiser.
- Closures. Supported, but each closure gets a unique type.
  `let c1 = |y| y + 1; let c2 = |z| z + 1; let foo = [c1, c2];` doesn't work.
  Solution: allocate closures, use references.
- Partial applications. Not supported, but we can use closures.
  Solution: generate both curried and uncurried versions.

Extraction to Rust: challenges

- Recursive data types: type definitions contain their fields by value.
  ```
  pub enum List<A> { nil, cons(A, List<A>) }
  ```
  has infinite size — invalid in Rust.
  Solution: use references:
  ```
  pub enum List<'a, A> {
    nil(PhantomData<&'a A>),
    cons(PhantomData<&'a A>, A, &'a List<'a, A>)
  }
  ```

- No GC. Currently, we use arena/region-based allocation.
  Possible to use existing GC, e.g. Boehm–Demers–Weiser.
- Closures. Supported, but each closure gets a unique type.
  `let c1 = |y| y + 1; let c2 = |z| z + 1; let foo = [c1, c2];` doesn't work.
  Solution: allocate closures, use references.
- Partial applications. Not supported, but we can use closures.
  Solution: generate both curried and uncurried versions.
- Inner (nested) fixpoints. Only top-level declarations are recursive.
  Solution: do recursion through the heap.

Extraction to Rust: `map`

Extracting `map : forall A B : Type, (A → B) → list A → list B`

```
impl<'a> Program {
 fn map<A: Copy, B: Copy>(&'a self, f: &'a dyn Fn(A) → B, l: &'a List<'a, A>)
       → &'a List<'a, B> {
   match l {
     &List::Nil(_) ⇒ { self.alloc(List::Nil(PhantomData))},
     &List::Cons(_, a, t) ⇒ {
       self.alloc(List::Cons(PhantomData, hint_app(f)(a), self.map(f, t)))
     },
   }
 }
 fn map__curried<A: Copy, B: Copy>(&'a self) → &'a dyn Fn(&'a dyn Fn(A) → B)
                → &'a dyn Fn(&'a List<'a, A>) → &'a List<'a, B> {
   self.closure(move |f| {
     self.closure(move |l| { self.map(f, l) })
   })}
 // ...
}
```

`Fn` — a standard function trait for closures

`hint_app` — a wrapper to guide type checking.

Extraction to Rust: smart contracts

- Smart contracts — programs "running on a blockchain".
- Rust is used as a smart contract language at Concordium.
- Additional challenge: integration with the target platform.
- We provide integration with the Concordium infrastructure.
- Generate wrappers, serialisation/deserialisation machinery, etc.
- Promising results: extracted code of reasonable size and performance.

Rust code:
`https://github.com/AU-COBRA/concordium-extracted-contracts`

Extraction to Elm

- Elm — general-purpose functional language.
- Based on Hindley-Milner type system (with some extensions).
- Used mostly for web development.
- Nice extraction target, but no unsafe type casts (like `Obj.magic`).
- Use pre-processing to inline/specialise definitions.
- Prenex-polymorphism + subset types = well-typed Elm code.

Extraction to Elm: Verified web app

- Inspired by Elm guide.
- An input form with user and password.
- Define the "logic" of the app in Coq.
- Use dependent types to encode model invariants.

Verified web app: the model

```
Record StoredEntry := { seName     : string;
                        sePassword : string }.

Definition ValidStoredEntry := { entry : StoredEntry | entry.(seName) ≠ ""
                                                     ∧ 8 ≤ length entry.(sePassword) }.
Record Model :=
  { (* A list of valid entries such with unique user names *)
    users : {l : list ValidStoredEntry | NoDup (seNames l)};
    (* A list of errors after validation *)
    errors : list string;
    (* Current user input *)
    currentEntry : Entry }.
```

Verified web app cont.

- Using valid entries — model invariants are preserved.
- No need for extra validation, apart from the entry point.
- Forces to think about validation.
- Guarantees that only valid entries are stored.

Verified web app: extraction

- Coq code uses `Program` for convenience.
- The resulting code is well-typed.
- We verify the main logic, extract, and append the views.
- The example is available in Ellie (link is in the abstract).

Future work

- Rust extraction:
  - Experiment with various GC.
  - More benchmarks.
  - Use `unsafe` to convince type checker, if required.
  - Connect ConCert to Rust semantics.
- Elm extraction:
  - Verify view $\longleftrightarrow$ model interaction.
  - The interaction model is quite similar to smart contract interaction in ConCert!

Thank you for your attention!

Our development:
`https://github.com/AU-COBRA/ConCert`

Add new extraction targets!