Validating OCaml soundness by translation into Coq

Jacques Garrigue, Takafumi Saikawa

Graduate School of Mathematics, Nagoya University

TYPES2022, 2022-Jun-23

Introduction

Starting point

- Proving the correctness of the full OCaml type inference is hard
  - Efforts exist to prove it part by part, but combining them is complex

Starting point

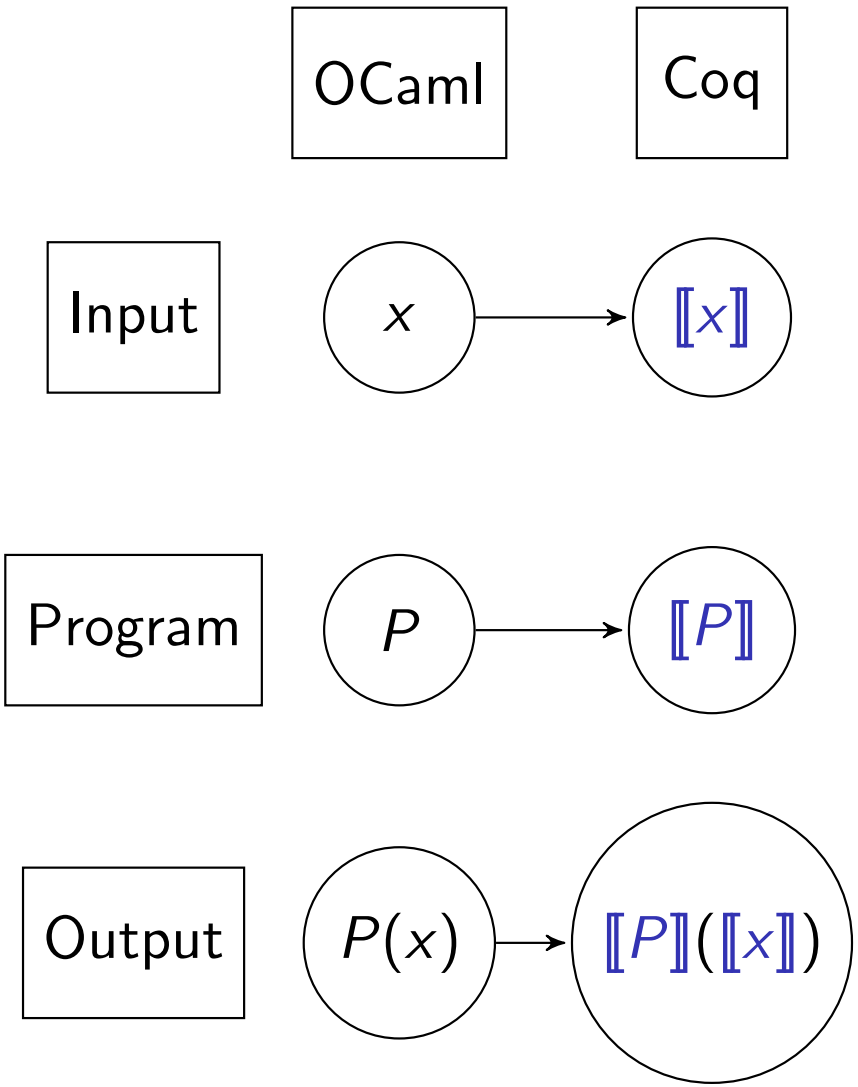
- Proving the correctness of the full OCaml type inference is hard
  - Efforts exist to prove it part by part, but combining them is complex
- Alternative approach: ensure that the generated typed syntax trees enjoys type soundness by translating them into another type system

```
ocamlc -c -coq foo.ml   ==>   foo.v
```

Soundness mantra

Typed programs must not be evaluated into wrongly typed values.

Soundness by translation



If for all $P : \tau \to \tau'$ and $x : \tau$

- P translates to $[\![P]\!]$, and $\vdash [\![P]\!] : [\![\tau \to \tau']\!]$
- x translates to $[\![x]\!]$, and $\vdash [\![x]\!] : [\![\tau]\!]$
- $[\![P]\!]$ applied to $[\![x]\!]$ evaluates to $[\![P(x)]\!]$
- $[\![\cdot]\!]$ is injective on types

then the soundness of Coq's type system implies the soundness of OCaml's evaluation

To evaluate translated programs

- No axioms in translated programs, so that the evaluation is not blocked
- Must implement OCaml's features, such as references, or polymorphic comparison inside Coq
- In turn this requires an intensional representation of OCaml's types, to be able to use them in computations

Translating types

Translating types

- Define a type representing OCaml types: `ml_type`
- And a translation function `coq_type : ml_type -> Type`
  This function must be computable.
- Wrap mutability and failure/non-termination into a monad

```
Definition M T := Env -> Env * (T + Exn).
```

- `Env` contains the state of reference cells.
  It is a mapping from keys (which contain some `T : ml_type`) to values of type `coq_type T`.
- `Exn` contains both ML exceptions and non-termination.
- Since `Env` and `Exn` may contain values of type `M T`, these definitions are mutually recursive, and need to bypass the positivity check.
- No other axiom or bypassing is used (at this point).

$$\texttt{ml\_type} : \texttt{Type}$$

$$\texttt{coq\_type}_M : \texttt{ml\_type} \to \texttt{Type}$$

Just like a Tarski universe, but M..

$$\texttt{ml\_type} : \texttt{Type}$$

$$\texttt{coq\_type}_M : \texttt{ml\_type} \to \texttt{Type}$$

Just like a Tarski universe, but M..

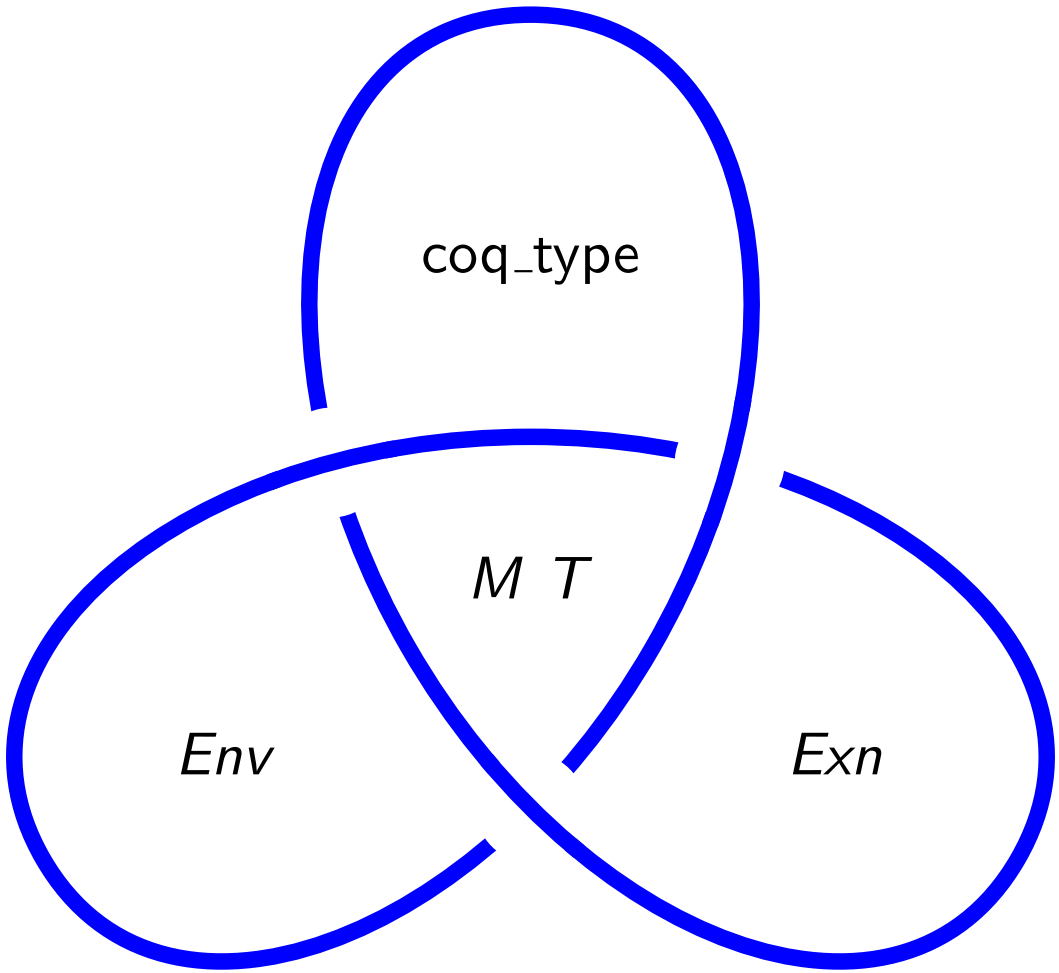
$$MT = \texttt{Env}_M \to \texttt{Env}_M \times (T + \texttt{Exn}_M)$$

$$\texttt{Env}_M = \texttt{int} \times \texttt{list binding}_M$$

$$\texttt{Exn}_M = (\text{exceptions})$$

$$\texttt{binding}_M = \texttt{int} \times (ty : \texttt{ml\_type}) \times (\texttt{coq\_type}\ ty)$$

Tying the Knot



Definition of `ml_type`

`ml_type` is just an inductive type with a branch for each OCaml type constructor used in the program. For instance:

```
Inductive ml_type :=
  | ml_int                    (* predefined types *)
  | ml_exn
  | ml_arrow (_ : ml_type) (_ : ml_type)
  | ml_ref (_ : ml_type)
  | ml_list (_ : ml_type)
  | ...
  | ml_color                  (* types from the program *)
  | ml_tree (_ : ml_type) (_ : ml_type)
  | ml_ref_vals (_ : ml_type).
```

Since it is used as a parameter for all polymorphic definitions, it needs to be defined first, but depends on nothing else.

Definition of coq_type

Once we have translated the type definitions, coq_types can be generated:

```
Variable M : Type -> Type.       (* The monad is not yet defined *)
Fixpoint coq_type (T : ml_type) : Type :=
  match T with
  | ml_int => Int63.int
  | ml_exn => ml_exns
  | ml_arrow T1 T2 => coq_type T1 -> M (coq_type T2)
  | ml_ref T1 => loc T1
  | ml_list T1 => list (coq_type T1)
  | ...
  | ml_color => color
  | ml_tree T1 T2 => tree (coq_type T1) (coq_type T2)
  | ml_ref_vals T1 => ref_vals (coq_type T1) T1
```

Thanks to this definition, polymorphic values need only take the intensional representation as parameter.

Building the execution monad

We can now build the monad, by relaxing one safeguard in Coq:

```
Record key := mkkey {key_id : int; key_type : ml_type}.
Record binding (M : Type -> Type) := mkbind
  { bind_key : key; bind_val : coq_type M (key_type bind_key) }.
Definition M0 Env Exn T := Env -> Env * (T + Exn).

#[bypass_check(positivity)]          (* non-positive definition *)
Inductive Env := mkEnv : int -> seq (binding (M0 Env Exn)) -> Env.
     with Exn := Catchable of coq_type (M0 Env Exn) ml_exn
               | GasExhausted | RefLookup | BoundedNat.

Definition M T := M0 Env T.
Definition Ret {A} (x : A) : M A := fun env => (env, inl x).
Definition Fail {A} (e : Exn) : M A := fun env => (env, inr e).
Definition Bind {A B} (x : M A) (f : A -> M B) : M B := ...
```

Translating programs

Translating recursive functions

To allow the translation of arbitrary recursive functions, all recursive functions take a gas parameter, and as a result may raise the exception `GasExhausted`.

```
let rec mccarthy_m n =                          (* pure arity = 1 *)
  if n > 100 then n - 10
  else mccarthy_m (mccarthy_m (n + 11));;

Fixpoint mccarthy_m (h : nat) (n : coq_type ml_int)
  : M (coq_type ml_int) :=
  if h is h.+1 then
    do v <- ml_gt h ml_int n 100%int63;      (* comparison *)
    if v then Ret (Int63.sub n 10%int63) else
      do v <- mccarthy_m h (Int63.add n 11%int63);
      mccarthy_m h v
  else Fail GasExhausted.
```

Comparison functions

OCaml allows polymorphic comparison. We mimic it by generating a type analyzing function.

```
Fixpoint compare_rec (h : nat) (T : ml_type)
  : coq_type T -> coq_type T -> M comparison :=
  if h is h.+1 then
    match T as T return coq_type T -> coq_type T -> M comparison with
    | ml_int => fun x y => Ret (Int63.compare x y)
    | ml_arrow T1 T2 =>                          (* fail as in OCaml *)
      fun x y => Fail (Catchable (Invalid_argument "compare"%string))
    | ml_ref T1 =>                  (* compare contents of references *)
      fun x y => compare_ref (compare_rec h) T1 x y
    | ml_ref_vals T1 => fun x y =>
        match x, y with RefVal x1 x2, RefVal y1 y2 =>
          lexi_compare (compare_rec h (ml_ref T1) x1 y1)
            (Delay (compare_rec h (ml_list T1) x2 y2))
        end
    ...
    end
  else fun _ _ => FailGas.
```

Breaking strong normalization...

Recall the seemingly innocuous non-positive definition of Env.

```
#[bypass_check(positivity)]          (* non-positive definition *)
Inductive Env := mkEnv : int -> seq (binding (M0 Env Exn)) -> Env.
```

This allows us to define really non-termination functions *without gas*...

Breaking strong normalization...

```
let omega x =
  let r = ref (fun x -> x) in
  let delta y = !r y in
  r := delta; delta x ;;
```

```
Definition omega (T : ml_type) (x : coq_type T) : M (coq_type T) :=
  do r <- newref (ml_arrow T T)
                 (fun x : coq_type T => Ret (x : coq_type T));
  let delta (y : coq_type T) : M (coq_type T) :=
    AppM (getref (ml_arrow T T) r) y in
  do _ <- setref (ml_arrow T T) r delta; delta x.
```

Note that one needs to use a reference, and this loop takes place only inside the monad. We believe that one cannot use this to prove `False` (, but not proved yet).

Simulating the toplevel

Contrary to C, OCaml allows toplevel statements (of pure arity 0) to change the global state. This is tricky to do this in Coq.

```
let r = ref [3] ;;
let z = r := 1 :: !r; !r;;
```

```
Definition Restart {A B} (x : W A) (f : M B) : W B :=
  BindW (fun _ => x) (fun _ => f).                (* W for Writer monad *)
Definition it : W unit := (empty_env, inl tt).

Definition r :=
  Restart it (newref (ml_list ml_int) (3%int63 :: @nil (coq_type ml_int))).
Definition z :=
  Restart r           (* the same state should only be restarted once! *)
    (do r <- FromW r;                (* can access the value repeatedly *)
     do _ <- (do v <- (do v <- getref (ml_list ml_int) r;
                         Ret (@cons (coq_type ml_int) 1%int63 v));
                setref (ml_list ml_int) r v);
     getref (ml_list ml_int) r).
Eval vm_compute in z.
```

Conclusion

Prospects

- Could also be used to do proofs about the translated programs, using the Monae library [Affeldt et al., 2019]
- We first plan to add our monad to the Monae hierarchy

- The use of an intentional representation for ML types should allow to properly translate GADTs
- Translating polymorphic variants and objects is another challenge

- Verification of the translator
- Theoretical account of the work

- Slides / poster at:

  `https://www.math.nagoya-u.ac.jp/~garrigue/cocti/`

Demo

Related work


Guillaume Claret. *Coq of OCaml*. OCaml Workshop, 2014.


Antal Spector-Zabusky *et al*. *Total Haskell is reasonable Coq*. CPP, 2018.


Danil Annenkov *et al*. *ConCert: a smart contract certification framework in Coq*. CPP, 2020.


Laila El-Beheiry *et al*. SMLtoCoq: Automated Generation of Coq Specifications and Proof Obligations from SML Programs with Contracts. LFMTP, 2021.


Matthieu Sozeau *et al*. *Coq Coq correct! verification of type checking and erasure for Coq, in Coq*, POPL, 2020.


Pierrick Couderc. *Vérification des résultats de l'inférence de types du langage OCaml.* PhD Thesis, Université Paris-Saclay, 2018.

Appendix

Translation of type definitions

- ML types have two representions in Coq: an intensional one as a term `t : ml_type`, and a shallow embedding `coq_type t`.
- In order to infer type equalities, some embedded types need to refer to intensional representations:

```
loc    : ml_type -> Type      (* translation of 'a ref *)
newref : forall (T : ml_type), coq_type T -> M (loc T)
```

- This creates a problem when translating polymorphic type definitions, as their type variables may be used either in an intensional or extensional way, and `coq_type` is not yet defined.
- Solution: use separate type parameters for intensional and extensional occurrences.

```
(* type 'a ref_vals = RefVal of 'a ref * 'a list *)
Inductive ref_vals (a : Type) (a_1 : ml_type) :=
  RefVal (_ : loc a_1) (_ : list a).
```

Purity analysis

- For each definition, we compute its *pure arity*, i.e. the number of applications before it may exhibit impure behavior.
- We use it to avoid turning all arrows into monadic ones.
- To avoid purity polymorphism, all function arguments are assumed to be values of pure arity 1.

```
type ('a,'b) tree =
  Leaf of 'a | Node of ('a,'b) tree * 'b * ('a,'b) tree ;;

let mknode t1 t2 = Node (t1, 0, t2) ;;          (* pure arity = 3 *)
```

```
Inductive tree (a : Type) (b : Type) :=
  | Leaf (_ : a)
  | Node (_ : tree a b) (_ : b) (_ : tree a b).

Definition mknode (T : ml_type) (t1 t2 : coq_type (ml_tree T ml_int))
  : coq_type (ml_tree T ml_int) :=
  Node (coq_type T) (coq_type ml_int) t1 0%int63 t2.
```

Handling GADTs

Our intensional representation of types allows to translate GADTs.
A simple approach is to embed equality proofs inside types.

```
type (_, _) eqw = Refl : ('a,'a) eqw
type empty = |
let cast : type t1 t2. (t1,t2) eqw -> t1 -> t2 = fun Refl x -> x
let int_not_empty : (int, empty) eqw -> empty = function _ -> .
```

```
Inductive eqw (T1 T2 : ml_type) := Refl of T1 = T2.
Inductive empty := .
Definition cast (T1 T2 : ml_type) (w : eqw T1 T2) (x : coq_type T1)
  : coq_type T2 :=
  match w with Refl H => eq_rect _ coq_type x _ H end.
Definition int_not_empty (x : eqw ml_int ml_empty) : empty.
  refine (match x with Refl H => _ end).
  discriminate.
Defined.
```

(Not implemented yet)

How to use

- New backend to OCaml, defined in the `ocaml_in_coq` branch of `COCTI/ocaml` on GitHub. (PR #3)

  `https://github.com/COCTI/ocaml/pull/3`

- Adds a `-coq` option to `ocamlc`, which switches to the Coq generation backend, producing a `.v` rather than a `.cmo`.
- At this point, supports only single file programs written in core ML plus references and algebraic datatypes (sum types), using a subset of `Pervasives`