Environment-friendly monadic equational reasoning for OCaml

Reynald Affeldt, Jacques Garrigue, Takafumi Saikawa

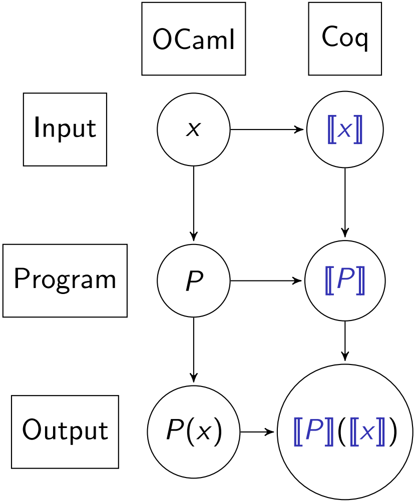
Graduate School of Mathematics, Nagoya University

TPP, October 30, 2023

Starting point : the Coqgen project

- Proving the correctness of the full OCaml type inference is hard
- We can prove it theoretically for subparts, but combining them is complex
- Writing a type checker for the typed syntax tree might help, but still suffers the same difficulties
- Alternative approach: ensure that the generated typed syntax trees enjoys type soundness by translating them into another type system, here Coq

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

Requirements for soundness

- Need to evaluate programs, so no axioms in translated programs
- Need to preserve Coq's soundness, so avoid other axioms too
- 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

Overview of translation

- 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).
```

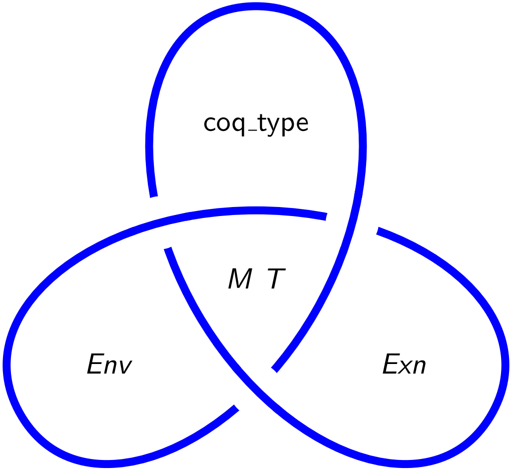
- `Env` contains the state of reference cells.
  It is a mapping from keys (which contain some `T : ml_type`) to values of type `coq_type T`.
- `Exn` contains both ML exceptions and non-termination.
- Since `Env` and `Exn` may contain values of type `M T`, these definitions are mutually recursive, and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

Type translation

The translation of types depends on the monad.

```
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_arrow T1 T2 => coq_type T1 -> M (coq_type T2)
  | ml_ref T1 => loc T1
  | ml_list T1 => list (coq_type T1)
  | ...
  end.
```

Tying the Knot



Status of Coqgen

Coqgen has been implemented as a backend to OCaml.
It is already able to translate many features

- Core ML : λ-calculus with polymorphism and recursion
- algebraic data types
- references and exceptions
- while and for loops
- lazy values
- etc...

It can be used as

- a soundness witness for type checking (as intended)
- a way to prove properties of programs, by translation $\Rightarrow$ this presentation

Monae

- Monae is a library for proving properties of programs using Monadic Equational Reasoning
- It already supports equational theories for many monads such as state, failure, probabilities and nondeterminism, and combinations of them.
- Soundness of reasoning is ensured by providing a model for the desired combination.
- Some of these models are provided as monad transformers, making it easy to build combinations.

Example: the array monad

The array monad describes an homogeneous store, with a default initial value.

```
HB.mixin Record isMonadArray (S : Type) (I : eqType) M of Monad M := {
  aget : I -> M S ;
  aput : I -> S -> M unit ;
  aputget : forall i s (A : Type) (k : S -> M A),
      aput i s >> aget i >>= k = aput i s >> k s ;
  aputC : forall i j u v, (i != j) \/ (u = v) ->
      aput i u >> aput j v = aput j v >> aput i u ; ... }.
```

Model, inheriting from the state monad.

```
Definition M := StateMonad.M (I -> S). (* the state is a function *)
Definition aget i : M S := fun a => (a i, a).
Definition insert i s (a : I -> S) j := if i == j then s else a j.
Definition aput i s : M unit := fun a => (tt, insert i s a). ...
HB.instance Definition _ := isMonadArray.Build
  S I M aputput aputget agetputskip agetget agetC aputC aputgetC.
```

The typed store monad

- A new monadic interface for Coqgen, allowing a heterogeneously typed store.
- Supports just references, but could be extended with exceptions and non-termination.
- Two models:
  - A full model, which mimicks exactly Coqgen, and as a result requires to bypass the positivity check.
  - A restricted model, which does not allow to put functions in the store, but is guaranteed to be sound.
    It corresponds to so-called *full-ground references* [KLMS17].

Basic operations (hierarchy.v)

```
Inductive loc (ml_type : Type) (locT : eqType) : ml_type -> Type :=
  mkloc T : locT -> loc locT T.
HB.mixin Structure isML_universe (ml_type : Type) := {
  eqclass : Equality.class_of ml_type ;
  coq_type : forall M : Type -> Type, ml_type -> Type ; ... }
#[short(type=ML_universe)]
HB.structure Definition ML_UNIVERSE := {ml_type & isML_universe ml_type}.
Canonical isML_universe_eqType (T : ML_universe) := EqType T eqclass.

HB.mixin Record isMonadTypedStore (MLU : ML_universe) (locT : eqType)
    (M : Type -> Type)  of Monad M := {
  cnew : forall {T : MLU}, coq_type M T -> M (loc locT T) ;
  cget : forall {T : MLU}, loc locT T -> M (coq_type M T) ;
  cput : forall {T : MLU}, loc locT T -> coq_type M T -> M unit ;
  crun : forall {A : Type}, M A -> option A ; (* execute in empty store *)
  ... }
```

Monadic laws

There are many laws, here are a few examples

```
cputget : forall T (r : loc locT T) (s : coq_type M T)
                 A (k : coq_type M T -> M A),
    cput r s >> (cget r >>= k) = cput r s >> k s ;
cnewget : forall T (s : coq_type M T) A (k : loc locT T -> coq_type M T -> M A),
    cnew s >>= (fun r => cget r >>= k r) = cnew s >>= (fun r => k r s) ;
cnewput : forall T (s t : coq_type M T) A (k : loc locT T -> M A),
    cnew s >>= (fun r => cput r t >> k r) = cnew t >>= k ;
cgetC : forall T1 T2 (r1 : loc locT T1) (r2 : loc locT T2)
               A (k : coq_type M T1 -> coq_type M T2 -> M A),
    cget r1 >>= (fun u => cget r2 >>= (fun v => k u v)) =
    cget r2 >>= (fun v => cget r1 >>= (fun u => k u v)) ;
```

Laws for crun

crun allows one to compare the result of computations by discarding the store.

```
crun : forall {A : Type}, M A -> option A ;
```

Note that the result type is an option. This is required so that we can build a model where store accesses are dynamically checked.

cput and cget may fail if a reference is undefined, or has a wrong type. Of course, this cannot happen if the translated program was well-typed.

```
crunret : forall (A B : Type) (m : M A) (s : B),
    crun m -> crun (m >> Ret s) = Some s ;
crunskip: crun skip = Some tt ;
crunnew : forall (A : Type) T (m : M A) (s : A -> coq_type M T),
    crun m -> crun (m >>= fun x => cnew (s x)) ;
```

Here the crun m condition means crun m <> None.

Commutation laws (typed_store_lib.v)

The above laws are insufficient to prove programs that use multiple references. We need to allow commutation.

```
cputnewC : forall T T' (r : loc locT T) (s : coq_type M T) (s' : coq_type M T')
                A (k : loc locT T' -> M A),
  cget r >> (cnew s' >>= fun r' => cput r s >> k r') = cput r s >> (cnew s' >>= k);
```

Here cget ensures that r exists before creating r', proving they are distinct.
It is convenient to introduce a derived operation cchk, which commutes with anything.

```
Definition cchk T (r : loc T) : M unit := cget r >> skip.
Lemma cnewchk T (s : coq_type M T) (A : Type) (k : loc T -> M A) :
  cnew s >>= (fun r => cchk r >> k r) = cnew s >>= k.
Lemma cchknewput T T' (r : loc T) (s : coq_type M T) (s' : coq_type M)
               A (k : loc locT T' -> M A) :
  cchk r >> (cnew s' >>= fun r' => cput r s >> k r') = cput r s >> (cnew s' >>= k).
```

Full ground model (monad_model.v)

In the full ground case, it is straightforward to build a model using the state monad transformer MS.

```
Record binding (M : Type -> Type) :=
  mkbind { bind_type : MLU; bind_val : coq_type M bind_type }.
Definition M : Type -> Type :=
  MS (seq (binding idfun)) [the monad of option_monad].
```

By passing the identity monad to binding we restrict the store to pure functions.

```
Let cnew T (v : coq_type M T) : M (loc T) := fun st =>
  let n := size st in Ret (mkloc T n, rcons st (mkbind T (v : coq_type' T))).
Let cget T (r : loc T) : M (coq_type M T) := fun st =>
  if nth_error st (loc_id r) is Some (mkbind T' v) then
    if coerce T v is Some u then Ret (u, st) else fail
  else fail.
Let crun (A : Type) (m : M A) : option A :=
  if m nil is (inr (a, _)) then Some a else None.
```

Recursively typed model (typed_store_model.v)

In the recursive case, we need to build an inductive type.

```
Record binding (MLU : ML_universe) (M : Type -> Type) :=
  mkbind { bind_type : MLU; bind_val : coq_type M bind_type }.

#[bypass_check(positivity)]
Inductive Env (MLU : ML_universe) :=
  mkEnv : seq (binding MLU (MS (Env MLU) option_monad)) -> Env MLU.

Definition M : Type -> Type := MS (Env MLU) option_monad.
```

The other definitions are essentially identical, but it still means that the model is proved in a setting where one can prove `False`.

Cyclic lists (cyclic.ml, cyclic.v, example_typed_store.v)

One can prove the standard example of separation logic using only our laws.

```
type 'a rlist = Nil | Cons of 'a * 'a rlist ref
let cycle a b =
  let r = ref Nil in let l = Cons (a, ref (Cons (b, r))) in
  r := l;    l
let hd x = function Nil -> x | Cons (a, _) -> a
```

translates to

```
Definition cycle (T : ml_type) (a b : coq_type T) : M (coq_type (ml_rlist T)) :=
  do r <- cnew (ml_rlist T) (Nil (coq_type T));
  do l <- (do v <- cnew (ml_rlist T) (Cons (coq_type T) b r);
          Ret (Cons (coq_type T) a v));
  do _ <- cput (ml_rlist T) r l; Ret l.
Definition hd (T : ml_type) (x : coq_type T) (param : coq_type (ml_rlist T))
  : coq_type T := match param with | Nil _ => x | Cons _ a _ => a end.
```

Cyclic lists (cont.)

```
Lemma hd_tl_tl_is_true :
  crun (do l <- cycle ml_bool true false; do l1 <- tl _ l; do l2 <- tl _ l1;
        Ret (hd ml_bool false l2)) = Some true.
Proof.
rewrite bindA -cnewchk.
under eq_bind => r1.
  under eq_bind do rewrite !bindA.
  under eq_bind do under eq_bind do rewrite !(bindA,bindretf) /=.
  under cchknewE do rewrite -bindA cputgetC //.
  rewrite cnewget /=.
  under eq_bind do under eq_bind do rewrite cputget /=.
  rewrite -bindA.
  over.
rewrite cnewchk -bindA crunret // -bindA_uncurry /= crungetput // bindA.
under eq_bind do rewrite !bindA.
under eq_bind do under eq_bind do rewrite bindretf /=.
by rewrite crungetnew // -(bindskipf (_ >>= _)) crunnewget // crunskip.
Qed.
```

Demo

Related work

- Coq-of-ocaml [GC14] and Hs-to-Coq [AS18] are also translators.
  - Explicitly geared at the proof of programs.
  - Neither comes with an equational theory.
- The typed-store monad is very close to Haskell's ST monad [LP94].
  - The latter additionaly uses polymorphism to scope references.
  - However, nobody seems to have developed laws for the ST monad.
- Staton and Kammar [KLMS17] have developed models for a typed store.
  - They only handle the full-ground case.
  - The store is statically typed, but it is not clear how one would handle lists of references for instance.
- At last, Sterling, Grazer and Birkedal [SGB23] have constructed a model allowing effectful functions in the store.
  - Their model uses a delay operation to avoid unguarded recursion.
  - It does not seem easily computable.

References

- Guillaume Claret. *Coq of OCaml*. OCaml Workshop, 2014.
- Antal Spector-Zabusky *et al*. *Total Haskell is reasonable Coq*. CPP, 2018.
- Jacques Garrigue and Takafumi Saikawa. *Validating OCaml soundness by translation into Coq*, TYPES, 2022.
- R. Affeldt, D. Nowak, T. Saikawa. *A hierarchy of monadic effects for program verification using equational reasoning*, MPC, 2019.
- R. Affeldt, D. Nowak. *Extending equational monadic reasoning with monad transformers*, TYPES, 2020.
- J. Launchbury, S. Peyton-Jones. *Lazy functional state threads*, PLDI, 1994.
- O. Kammar, P. B. Levy, S. K. Moss, S. Staton. *A monad for full ground reference cells*, LICS, 2017.
- J. Sterling, D. Gratzer, L. Birkedal. *Denotational semantics of general store and polymorphism*, 2023.

Thank you

For more information see

http://www.math.nagoya-u.ac.jp/~garrigue/cocti/coqgen/