Environment-friendly monadic equational reasoning for OCaml

Reynald Affeldt, Jacques Garrigue, Takafumi Saikawa

Graduate School of Mathematics, Nagoya University

NIER, November 26, 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

$$\begin{tikzcd}
 & \boxed{\text{OCaml}} & \boxed{\text{Coq}} \\
\boxed{\text{Input}} & x \arrow[r] \arrow[d] & \llbracket x \rrbracket \arrow[d] \\
\boxed{\text{Program}} & P \arrow[r] \arrow[d] & \llbracket P \rrbracket \arrow[d] \\
\boxed{\text{Output}} & P(x) \arrow[r] & \llbracket P \rrbracket(\llbracket x \rrbracket)
\end{tikzcd}$$

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

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

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

Overview of translation

- Define a type representing OCaml types: `ml_type`
  (needed for building a dynamically typed store)
- 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.

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.
```

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 ⇒ 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 (k : S -> M A),
    aput i s >> aget i >>= k = aput i s >> k s ;
  aputgetC : forall i j u A (k : S -> M A), i != j ->
    aput i u >> aget j >>= k = aget j >>= (fun v => aput i u >> k v) ; ... }.
```

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 ...
```

Building a new monad bottom-up

Usually, one starts from a well-established equational theory.

The ability to prove interactively within Coq offers a new bottom-up methodology.

1. Define interface operations
2. Define a model for these operations
3. Add/modify laws in the interface
4. Prove the laws with the model
5. Try proving some program using the laws
6. Succeed, or go back to step 3

The typed store monad (`hierarchy.v`)

- Focus on the use of references in ML.
- Operations are the same as Haskell's ST monad.

```
cnew : forall {T : ml_type}, coq_type N T -> M (loc T)
cget : forall {T : ml_type}, loc T -> M (coq_type N T)
cput : forall {T : ml_type}, loc T -> coq_type N T -> M unit
crun : forall {A : Type}, M A -> option A ; (* execute in empty store *)
```

  Unfortunately, no equational theory is known for the ST monad.
- Start from the Array monad, and add laws for `cnew`.
- Need failure in the model, for dynamically typed access to the store.
  Hence `crun` returns an option type.

Laws for `cnew`

The basic laws are similar to `aput`.

```
cnewget : cnew s >>= (fun r => cget r >>= k r)  = cnew s >>= (fun r => k r s)
cnewput : cnew s >>= (fun r => cput r t >> k r) = cnew t >>= k
```

Problem: how can we allow commuting `cnew` with other operations, without introducing a notion of freshness?

```
cputnewC : cput r s >> (cnew s' >>= k) = ??
```

Intuition: since `r` is valid before creating the new reference, the two operations should commute.

Asserting validity of a reference with `cchk`

Our solution is to add new operation `cchk r`, which ensures that

- there is a value in the store corresponding to the reference `r`,
- and this value has the right type.

By adding a `cchk` before `cnew` we can ensure that `loc_id r` $\neq$ `loc_id r'`.

```
cputnewC : cput r s >> (cnew s' >>= k) =
           cchk r >> (cnew s' >>= fun r' => cput r s >> k r')
cchknewE : (* generate inequation *)
  (forall r2 : loc T2, loc_id r1 != loc_id r2 -> k1 r2 = k2 r2) ->
  cchk r1 >> (cnew T2 s >>= k1) = cchk r1 >> (cnew T2 s >>= k2)
```

Remark: actually, we can pose

```
Definition cchk {T} (r : loc T) := cget r >> skip.
```

Example: commutation at a distance

```
Lemma perm3 T (s1 s2 s3 s4 : coq_type N T) :
  do r1 <- cnew s1; do r2 <- cnew s2; do r3 <- cnew s3; cput r1 s4 =
  do r1 <- cnew s4; do r2 <- cnew s2; do r3 <- cnew s3; skip :> M _.
Proof.
```

$$\mathtt{cnew}\ s_1 \gg\!= \lambda r_1.\mathtt{cnew}\ s_2 \gg \mathtt{cnew}\ s_3 \gg \mathtt{cput}\ r_1\ s_4$$

```
rewrite -cnewchk. (* introduce cchk *)
```

$$\mathtt{cnew}\ s_1 \gg\!= \lambda r_1.\underline{\mathtt{cchk}\ r_1} \gg \mathtt{cnew}\ s_2 \gg \mathtt{cnew}\ s_3 \gg \mathtt{cput}\ r_1\ s_4$$

```
under eq_bind do rewrite -cchknewC. (* commute under binder *)
```

$$\mathtt{cnew}\ s_1 \gg\!= \lambda r_1.\mathtt{cchk}\ r_1 \gg \mathtt{cnew}\ s_2 \gg \underline{\mathtt{cchk}\ r_1} \gg \mathtt{cnew}\ s_3 \gg \mathtt{cput}\ r_1\ s_4$$

```
under eq_bind do rewrite -[cput _ _]bindmskip. (* add skip after cput *)
```

$$\mathtt{cnew}\ s_1 \gg\!= \lambda r_1.\mathtt{cchk}\ r_1 \gg \mathtt{cnew}\ s_2 \gg \mathtt{cchk}\ r_1 \gg \mathtt{cnew}\ s_3 \gg \mathtt{cput}\ r_1\ s_4 \gg \underline{\mathtt{skip}}$$

```
under eq_bind do rewrite -2!cputnewC. (* commute twice *)
```

$$\mathtt{cnew}\ s_1 \gg\!= \lambda r_1.\underline{\mathtt{cput}\ r_1\ s_4} \gg \mathtt{cnew}\ s_2 \gg \mathtt{cnew}\ s_3 \gg \mathtt{skip}$$

```
rewrite cnewput. (* update state *)
```

$$\mathtt{cnew}\ \underline{s_4} \gg\!= \lambda r_1.\mathtt{cnew}\ s_2 \gg \mathtt{cnew}\ s_3 \gg \mathtt{skip}$$

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.

```
crunskip : crun skip = Some tt ;
crunret  : crun m -> crun (m >> Ret s) = Some s ;
crunnew  : crun m -> crun (m >>= fun x => cnew (s x)) ;
```

Here the `crun m` condition means `crun m` $\neq$ None, i.e. `m` does not fail.

Full ground model (`monad_model.v`)

We can build a model using the state monad transformer `MS`.
This covers the full ground case [KLMS17], i.e., no side-effecting functions in the store.

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

By passing a distinct monad `N` to `coq_type` we restrict the store to pure functions.

```
Let cnew T (v : coq_type N 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 N 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.
```

Cyclic lists (`cycle.ml`, `cycle.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
let tl = function Nil -> Nil | Cons (_, l) -> !l
```

translates to

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

$$(\mathtt{cnew}\ \mathtt{Nil} \gg= \lambda r.(\mathtt{cnew}\ (\mathtt{Cons}\ \mathtt{f}\ r) \gg= \lambda v.\mathtt{ret}\ (\mathtt{Cons}\ \mathtt{t}\ v)) \gg= \lambda l.\mathtt{cput}\ r\ l \gg \mathtt{ret}\ l)$$
$$\gg= \lambda l.\mathtt{tl}\ l \gg= \lambda l_1.\mathtt{tl}\ l_1 \gg= \lambda l_2.\mathtt{ret}\ (\mathtt{hd}\ \mathtt{f}\ l_2)$$

```
rewrite bindA -cnewchk. (* insert cchk *)
```

$$\mathtt{cnew}\ \mathtt{Nil} \gg= \lambda r.\underline{\mathtt{cchk}\ r} \gg ((\mathtt{cnew}\ \ldots \gg= \lambda v.\mathtt{ret}\ (\mathtt{Cons}\ \mathtt{t}\ v)) \gg= \lambda l.\mathtt{cput}\ r\ l \gg \mathtt{ret}\ l)$$
$$\gg= \lambda l.\mathtt{tl}\ l \gg= \lambda l_1.\mathtt{tl}\ l_1 \gg= \lambda l_2.\mathtt{ret}\ (\mathtt{hd}\ \mathtt{f}\ l_2)$$

```
under eq_bind => r1. (* go under binders *)
under eq_bind do rewrite !bindA.
under cchknewE => r2 r1r2. (* deduce r1r2 from cchk >> cnew *)
```

```
r1r2 : loc_id r1 != loc_id r2
```

$$(\mathtt{ret}\ (\mathtt{Cons}\ \mathtt{t}\ r_2) \gg= \lambda l.\mathtt{cput}\ r_1\ l \gg \mathtt{ret}\ l) \gg= \lambda l.\mathtt{tl}\ l \gg= \lambda l_1.\mathtt{tl}\ l_1 \gg= \lambda l_2.\mathtt{ret}\ (\mathtt{hd}\ \mathtt{f}\ l_2)$$

```
rewrite !(bindA,bindretf) -bindA. (* substitutions *)
```

$$(\mathtt{cput}\ r_1\ \underline{(\mathtt{Cons}\ \mathtt{t}\ r_2)} \gg \underline{\mathtt{tl}\ (\mathtt{Cons}\ \mathtt{t}\ r_2)}) \gg= \lambda l_1.\mathtt{tl}\ l_1 \gg= \lambda l_2.\mathtt{ret}\ (\mathtt{hd}\ \mathtt{f}\ l_2)$$

```
rewrite /=. (* simplify *)
```

$$(\mathtt{cput}\ r_1\ (\mathtt{Cons}\ \mathtt{t}\ r_2) \gg \underline{\mathtt{cget}\ r_2}) \gg= \lambda l_1.\mathtt{tl}\ l_1 \gg= \lambda l_2.\mathtt{ret}\ (\mathtt{hd}\ \mathtt{f}\ l_2)$$

```
rewrite cputgetC //. (* uses r1r2 *)
```

$$\mathtt{cget}\ r_2 \gg= \lambda v.\mathtt{cput}\ r_1\ (\mathtt{Cons}\ \mathtt{t}\ r_2) \gg \mathtt{tl}\ v \gg= \lambda l_2.\mathtt{ret}\ (\mathtt{hd}\ \mathtt{f}\ l_2)$$

```
over. (* leave cchknewE *)
```

$$\mathsf{cchk}\ r_1 \gg \mathsf{cnew}\ (\mathtt{Cons\ f}\ r_1) \mathrel{\gg\!=} \lambda r_2.\mathsf{cget}\ r_2 \mathrel{\gg\!=} \lambda v.\mathsf{cput}\ r_1\ (\mathtt{Cons\ t}\ r_2) \\ \gg \mathtt{tl}\ v \mathrel{\gg\!=} \lambda l_2.\mathtt{ret}\ (\mathsf{hd}\ \mathtt{f}\ l_2)$$

```
rewrite cnewget.
```

$$\mathsf{cchk}\ r_1 \gg \mathsf{cnew}\ (\mathtt{Cons\ f}\ r_1) \mathrel{\gg\!=} \lambda r_2.\mathsf{cput}\ r_1\ (\mathtt{Cons\ t}\ r_2) \\ \gg \mathtt{tl}\ (\mathtt{Cons\ f}\ r_1) \mathrel{\gg\!=} \lambda l_2.\mathtt{ret}\ (\mathsf{hd}\ \mathtt{f}\ l_2)$$

```
rewrite /=.
```

$$\mathsf{cchk}\ r_1 \gg \mathsf{cnew}\ (\mathtt{Cons\ f}\ r_1) \mathrel{\gg\!=} \lambda r_2.\mathsf{cput}\ r_1\ (\mathtt{Cons\ t}\ r_2) \gg \mathsf{get}\ r_1 \mathrel{\gg\!=} \lambda l_2.\mathtt{ret}\ (\mathsf{hd}\ \mathtt{f}\ l_2)$$

```
under cchknewE do rewrite cputget.
```

$$\mathsf{cchk}\ r_1 \gg \mathsf{cnew}\ (\mathtt{Cons\ f}\ r_1) \mathrel{\gg\!=} \lambda r_2.\mathsf{cput}\ r_1\ (\mathtt{Cons\ t}\ r_2) \gg \lambda l_2.\mathtt{ret}\ (\mathsf{hd}\ \mathtt{f}\ (\mathtt{Cons\ t}\ r_2))$$

```
rewrite /=.
```

$$\mathsf{cchk}\ r_1 \gg \mathsf{cnew}\ (\mathtt{Cons\ f}\ r_1) \mathrel{\gg\!=} \lambda r_2.\mathsf{cput}\ r_1\ (\mathtt{Cons\ t}\ r_2) \gg \lambda l_2.\mathtt{ret\ t}$$

```
over. (* leave binder *)
rewrite cnewchk. (* remove cchk *)
```

$$\mathsf{cnew}\ \mathtt{Nil} \mathrel{\gg\!=} \lambda r_1.\mathsf{cnew}\ (\mathtt{Cons\ f}\ r_1) \mathrel{\gg\!=} \lambda r_2.\mathsf{cput}\ r_1\ (\mathtt{Cons\ t}\ r_2) \gg \lambda l_2.\mathtt{ret\ t}$$

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/