Mechanized monadic equational reasoning for ML references

Reynald Affeldt, Jacques Garrigue, Takafumi Saikawa

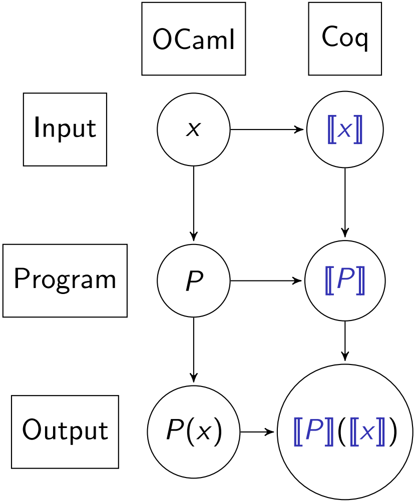
Graduate School of Mathematics, Nagoya University

CAMBIUM, 13 septembre 2024

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

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 -> option (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 ML exceptions.
- `option` is for type errors and non-termination.

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 *)
cref   : 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.

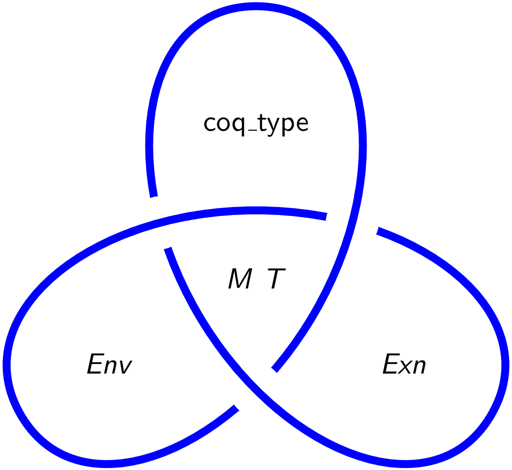
```
(* type 'a ref_vals = RefVal of 'a ref * 'a list *)
Inductive ref_vals (a : Type) (a_1 : ml_type) :=
  RefVal (_ : loc a_1) (_ : list a).
```

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          => PrimInt63.int
  | ml_arrow T1 T2 => coq_type T1 -> M (coq_type T2)
  | ml_ref T1       => loc T1                      (* Type of references *)
  | ml_list T1      => list (coq_type T1)
  | ...
  end.
```

Tying the Knot



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.
```

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.
```

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 (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 laws to 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
```

  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.

† crun is part of the typed-store-run monad.

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) option_monad.
```

By using a distinct monad N in binding we prevent functions on M in the store.

```
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)).
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.                        (* correctly translated code never fails *)
Let crun (A : Type) (m : M A) : option A :=
  if m nil is (inr (a, _)) then Some a else None.
```

Higher-order model (typed_store_model.v)

If we want to translate arbitrary OCaml code, we need lift this restriction.
This can be done by making the store (non-positively) inductive, so that $N = M$.

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

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

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

The other definitions are essentially identical.

Higher-order model (typed_store_model.v)

If we want to translate arbitrary OCaml code, we need lift this restriction.
This can be done by making the store (non-positively) inductive, so that N = M.

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

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

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

The other definitions are essentially identical.

Unfortunately this model allows to prove `False`.
Do not know yet whether the associated equational theory allows to prove `False` too.

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
```

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 r1` $\neq$ `loc_id r2`.

```
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)

cchknewput : cchk r1 >> (cnew s' >>= fun r2 => cput r1 s >> k r2)
           = cput r1 s >> (cnew s' >>= k)
```

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 r1` $\neq$ `loc_id r2`.

```
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)

cchknewput : cchk r1 >> (cnew s' >>= fun r2 => cput r1 s >> k r2)
           = cput r1 s >> (cnew s' >>= k)
```

Remark: actually, we can pose

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

Other laws

```
cgetput : cget r >> cput r s = cput r s ;   cgetputskip : cget r >>= cput r = cget r >> skip ;
cgetget : cget r >>= (fun s => cget r >>= k s) = cget r >>= fun s => k s s ;
cputget : cput r s >> (cget r >>= k) = cput r s >> k s ;
cputput : cput r s >> cput r s' = cput r s' ;
cgetC : cget r1 >>= (fun u => cget r2 >>= (fun v => k u v)) = cget r2 >>= (fun v => cget r1 >>= (f
cgetputC : cget r1 >> cput r2 s = cput r2 s >> cget r1 >> skip ;
cputC : loc_id r1 != loc_id r2 \/ JMeq s1 s2 -> cput r1 s1 >> cput r2 s2 = cput r2 s2 >> cput r1 s
cputgetC : loc_id r1 != loc_id r2 ->
           cput r1 s1 >> cget r2 >>= k = cget r2 >>= (fun v => cput r1 s1 >> k v) ;
cnewchk : cnew T s >>= (fun r => cchk r >> k r) = cnew T s >>= k ;
cchknewC : cchk r >> (cnew T2 s >>= fun r' => cchk r >> k r') = cchk r >> (cnew T2 s >>= k) ;
cchkgetC : cchk r1 >> (cget r2 >>= k) = cget r2 >>= (fun s => cchk r1 >> k s) ;
cchknewget : cchk r >> (cnew T' s >>= fun r' => cget r >>= k r') = cget r >>= (fun u => cnew T' s
cchkget : cchk r >> (cget r >>= k) = cget r >>= k ;
cgetchk : cget r >>= (fun s => cchk r >> k s) = cget r >>= k ;
cchkputC : cchk r1 >> cput r2 s = cput r2 s >> cchk r1 ;
cchkput : cchk r >> cput r s = cput r s ;           cputchk : cput r s >> cchk r = cput r s ;
cchkC : cchk r1 >> cchk r2 = cchk r2 >> cchk r1 ;   cchkdup : cchk r >> cchk r = cchk r ;
```

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.
```

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

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

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

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

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

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

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

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

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

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

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

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 rtl (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 rtl_tl_self T (a b : coq_type N T) :
  do l <- cycle T a b; do l1 <- rtl l; rtl l1 = cycle T a b.
Proof.
rewrite /cycle bindA -[LHS]cnewchk.
under eq_bind => r1.
  rewrite bindA; under eq_bind do rewrite !bindA.
  under cchknewE do
    rewrite bindretf bindA bindretf bindA cputget bindretf -bindA cputgetC //.
  rewrite cnewget; over.
rewrite cnewchk.
by under [RHS]eq_bind do (rewrite bindA; under eq_bind do rewrite bindretf).
Qed.
```

Proof of rtl_tl_self

$$(\mathtt{cnew\ Nil} \gg= \lambda r.(\mathtt{cnew\ (Cons\ b}\ r) \gg= \lambda v.\mathtt{Ret\ (Cons\ a}\ v)) \gg= \lambda l.\mathtt{cput}\ r\ l \gg \mathtt{Ret}\ l) \gg= \lambda l.\mathtt{rtl}\ l \gg= \mathtt{rtl}$$

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

$$\mathtt{cnew\ Nil} \gg= \lambda r.\underline{\mathtt{cchk}\ r} \gg ((\mathtt{cnew}\ \ldots \gg= \lambda v.\mathtt{Ret\ (Cons\ a}\ v)) \gg= \lambda l.\mathtt{cput}\ r\ l \gg \mathtt{Ret}\ l) \gg=\gg= \lambda l.\mathtt{rtl}\ l \gg= \mathtt{rtl}$$

```
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`

$$(\mathtt{Ret\ (Cons\ a}\ r_2) \gg= \lambda l.\mathtt{cput}\ r_1\ l \gg \mathtt{Ret}\ l) \gg= \lambda l.\mathtt{rtl}\ l \gg= \mathtt{rtl}$$

```
rewrite bindretf bindA bindretf. (* substitutions *)
```

$$\mathtt{cput}\ r_1\ \underline{(\mathtt{Cons\ a}\ r_2)} \gg (\underline{\mathtt{rtl}\ r1} \gg \mathtt{rtl})$$

```
rewrite bindA cputget
```

cput r_1 (Cons a r_2) $\gg$ ($\underline{\text{Ret } r_2}$ $\gg$ rtl)

```
rewrite bindretf.
```

cput r_1 (Cons a r_2) $\gg$ rtl r_2

```
rewrite -bindA cputgetC //; over. (* use r1r2 and leave cchknewE *)
```

cchk r_1 $\gg$ (cnew (Cons b r_1) $\gg=$ λr_2.cget r_2 $\gg=$ λv.cput r_1 (Cons a r_2)
$\gg$ Ret (match v with Nil $\Rightarrow$ r_2 | Cons _ t $\Rightarrow$ t end))

```
rewrite cnewget.
```

cchk r_1 $\gg$ (cnew (Cons b r_1) $\gg=$ λr_2.cput r_1 (Cons a r_2) $\gg$ Ret r_1)

```
over. (* leave binder *)
```

cnew Nil $\gg=$ λr_1.cchk r_1 $\gg$ (cnew (Cons b r_1) $\gg=$ λr_2.cput r_1 (Cons a r_2) $\gg$ Ret r_1)

```
rewrite cnewchk.
```

cnew Nil $\gg=$ λr_1.cnew (Cons b r_1) $\gg=$ λr_2.cput r_1 (Cons a r_2) $\gg$ Ret r_1

Laws for `crun`

`crun` allows one to compare the result of computations by discarding the store.
Useful to prove equivalence between imperative and functional algorithms.

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

The result type is an option, to allow failure in the model.
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)) ;
(* and three more laws: crunnewgetC, crungetput, crunmskip *)
```

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

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/