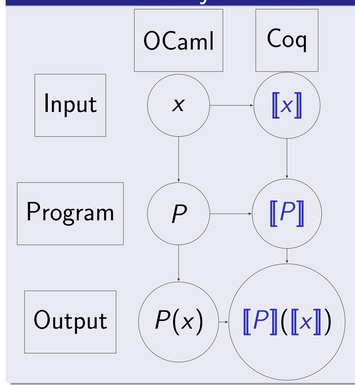
A Gallina generating backend to check OCaml's type inference correctness

Jacques Garrigue, Takafumi Saikawa (Graduate School of Mathematics, Nagoya University)

1. Starting point

- 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 enjoy type soundness by translating them into another type system

2. 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

3. 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

Challenge 1: Bootstrapping the OCaml World

- We need to define a monad to handle OCaml effects
- But monadic values may appear in the store or inside exceptions, which are wrapped in the monad
- Moreover, the type of those values is a function of their intensional type

4. Solution

Define

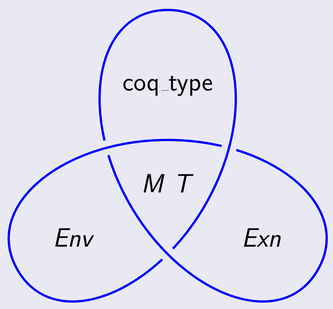
- Env (the store),
- Exn (exceptional results), and
- $M\ T = Env \to Env \times (T + Env)$

by mutual recursion.

This requires breaking positivity.

Parameterize them on both

- `ml_type` (the type of intensional ML types), and
- `coq_type` (the function translating them to Coq types)



Note that the latter is itself paramerized on the monad, but only in an abstract way, so that we can tie the knot.

See `https://www.math.nagoya-u.ac.jp/~garrigue/cocti/index.html` and `https://github.com/COCTI/ocaml/pull/3` for more information and a working compiler. COCTI is supported by the Tezos Foundation.

5. The REFmonad functor

```
Module Type MLTY.
  Parameter ml_type : Set.
  Parameter ml_exn : ml_type.
  Parameter coq_type : forall M : Type -> Type, ml_type -> Type.
End MLTY.
Module REFmonad(MLtypes : MLTY).
  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 := Env -> Env * (T + Exn).          (* = M0 Env Exn 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 := ...
  ...                              (* monadic operations for references *)
End REFmonad.
```

Challenge 2: Translating type definitions

The type definitions of a program are used to generate 3 distinct outputs:

- An intensional type representation: `ml_type`
- Concrete type definitions
- A translation function `coq_type : ml_type -> Type`, from the intensional represention to the concrete one, which must be computable.

Concrete definitions may mention intensional types, and `coq_type` clearly requires both. As a result, concrete definitions cannot use `coq_type`.

6. OCaml source types

```
type color = Red | Green | Blue
type 'a endo = Endo of ('a -> 'a)
type 'a ref_vals = RefVal of 'a ref * 'a list
```

7. Coq output

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

Module MLtypes.
Record key := mkkey {key_id : int; key_type : ml_type}.
Inductive loc : ml_type -> Type :=            (* the type of references *)
  mkloc : forall k : key, loc (key_type k).

Section with_monad.
Variable M : Type -> Type.

Inductive color := Red | Green | Blue.
Inductive endo (a : Type) := Endo (_ : a -> M a).
Inductive ref_vals (a : Type) (a_1 : ml_type) :=
  RefVal (_ : loc a_1) (_ : list a).          (* need both a and a_1 *)
Inductive ml_exns :=
  Invalid_argument (_ : string) | Failure (_ : string) | Not_found.
```

8. Coq output / Applying the functor

```
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            (* use the intensional representation *)
  | ml_list T1 => list (coq_type T1)
  ...
  | ml_color => color
  | ml_endo T1 => endo (coq_type T1)
  | ml_ref_vals T1 => ref_vals (coq_type T1) T1 (* a and a1 are both T1 *)
  end.
End with_monad.
End MLtypes. Export MLtypes.

(* Apply the functor to our types *)
Module REFmonadML := REFmonad (MLtypes). Export REFmonadML.
```

Challenge 3: Translating programs

Now that we have defined our types and monad, we can translate programs. There remains 3 problems.

- How to translate polymorphism and avoid nested monadification.
- How to translate recursion.
- How to thread the state in programs.

9. Purity analysis and polymorphic functions

For each definition, we compute its *pure arity*, i.e. the number of applications before it may exhibit impure behavior. Function arguments are assumed to have pure arity 1. Polymorphic functions are intensional (they use `ml_type`).

```
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.
```

10. Translating recursive functions

All recursive functions take a gas parameter, and may raise `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 : int) : M 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.
```

11. 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] ;;

Definition Restart {A B} (x : W A) (f : M B) : W B := f (fst x).
Definition it : W unit := (empty_env, inl tt).
Definition r : W (loc (ml_list ml_int)) :=
  Restart it (newref (ml_list ml_int) (3%int63 :: @nil (coq_type ml_int))).
```