

Environment-friendly monadic equational reasoning for OCaml

Reynald Affeldt Jacques Garrigue Takafumi Saikawa

ITP 2023, Białystok

Outline

Overview

Coq semantics of OCAML types

Monadic Semantics of OCAML Programs

Examples

Conclusions

This presentation

- ▶ Goal: We want to do equational reasoning on OCAML programs
- ▶ Approach: reuse¹ the output of COQGEN (OCAML \rightarrow COQ)
 - ▶ COQGEN encapsulates effects into a monad;
we therefore want to use *monadic* equational reasoning
 - ▶ we want to keep OCAML programs executable in COQ
- ▶ Contributions:
 - ▶ equational theory to reason about OCAML programs
 - ▶ verification library (design interface + lemmas)
 - ▶ concrete, COQ-executable examples

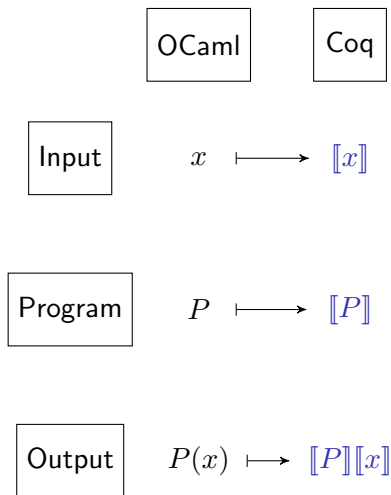
¹thus environment-friendly...

Building on previous work

This work relies on the following components:

- ▶ SSREFLECT
 - ▶ In particular, its rewriting tactic and the `under` tactical
- ▶ MONAE [Affeldt et al., 2019]
 - ▶ Hierarchy of monad interfaces + models + applications
 - ▶ Which relies on HIERARCHY-BUILDER [Cohen et al., 2020]
- ▶ COQGEN [Garrigue and Saikawa, 2022]
 - ▶ `ocamlc -c -coq`
 - ▶ monadic shallow embedding of OCAML programs into Coq

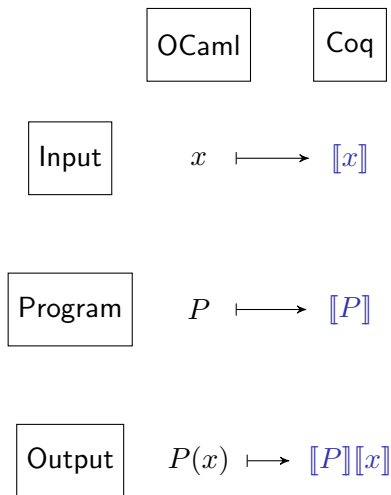
Soundness by translation [Garrigue and Saikawa, 2022]



For function $P : \tau \rightarrow \tau'$ and input $x : \tau$

- ▶ P translates to $\llbracket P \rrbracket$, and $\vdash \llbracket P \rrbracket : \llbracket \tau \rightarrow \tau' \rrbracket$
- ▶ x translates to $\llbracket x \rrbracket$, and $\vdash \llbracket x \rrbracket : \llbracket \tau \rrbracket$
- ▶ run $\llbracket P \rrbracket \llbracket x \rrbracket$ in COQ to check
 1. it evaluates to $\llbracket P(x) \rrbracket$
 2. it is typed as $\vdash \llbracket P(x) \rrbracket : \llbracket \tau' \rrbracket$

Soundness by translation [Garrigue and Saikawa, 2022]



For function $P : \tau \rightarrow \tau'$ and input $x : \tau$

- ▶ P translates to $\llbracket P \rrbracket$, and $\vdash \llbracket P \rrbracket : \llbracket \tau \rightarrow \tau' \rrbracket$
- ▶ x translates to $\llbracket x \rrbracket$, and $\vdash \llbracket x \rrbracket : \llbracket \tau \rrbracket$
- ▶ run $\llbracket P \rrbracket \llbracket x \rrbracket$ in Coq to check
 1. it evaluates to $\llbracket P(x) \rrbracket$
 2. it is typed as $\vdash \llbracket P(x) \rrbracket : \llbracket \tau' \rrbracket$

Executability is a design principle of COQGEN

Example: translation of a pure function

OCAML (pure.ml)

```
let discriminant a b c = b * b - 4 * a * c
```

⇓ `ocamlc -c -coq`

Coq

```
Definition discriminant (a b c : coq_type ml_int)
  : coq_type ml_int :=
  PrimInt63.sub (PrimInt63.mul b b)
  (PrimInt63.mul (PrimInt63.mul 4%int63 a) c).
```

- ▶ `ml_int` is a deep-embedding of the OCAML type `int` and `(coq_type ml_int)` is its interpretation in COQ.

Outline

Overview

Coq semantics of OCAML types

Monadic Semantics of OCAML Programs

Examples

Conclusions

Translation of types

OCAML

(primitive)

`int`, `bool`, ...

(function)

`t0 → t1`

(reference)

`t ref`

(user-defined)

`type 'a rlist =`

| `Nil`

| `Cons of`

`'a * 'a rlist ref`

CoQ

```
Inductive ml_type :=
```

```
| ml_int | ml_bool | ...
```

```
| ml_arrow : ml_type -> ml_type -> ml_type
```

```
| ml_ref    : ml_type -> ml_type
```

```
| ml_rlist  : ml_type -> ml_type.
```

```
Variant loc (ml_type:Type) (locT:eqType)
```

```
: ml_type -> Type :=
```

```
mkloc T : locT -> loc locT T.
```

```
Inductive rlist (a : Type) (a_1 : ml_type) :=
```

```
| Nil
```

```
| Cons : a -> loc (ml_rlist a_1) -> rlist a a_1.
```

- ▶ `ml_type` is a deep-embedding of the syntax of OCAML types
- ▶ `loc` and `rlist` are auxiliary types for the semantics

Translation of types – interpretation

reminder

```
Variant loc (ml_type:Type) (locT:eqType)
  : ml_type -> Type :=
  mkloc T : locT -> loc locT T.
```

```
Inductive rlist (a : Type) (a_1 : ml_type) :=
  | Nil
  | Cons : a -> loc (ml_rlist a_1) -> rlist a a_1.
```

```
Fixpoint coq_type63 (M : Type -> Type) (T : ml_type) : Type :=
  match T with
  | ml_int => int
  | ml_bool => bool
  | ml_unit => unit
  | ml_arrow T1 T2 => coq_type63 T1 -> M (coq_type63 T2)
  | ml_ref T1 => loc T1
  | ml_rlist T1 => rlist (coq_type63 T1) T1
  end.
```

- ▶ References need both the syntax and interpretation of a type
- ▶ Functions may have effects (M at the codomain)

Packing syntactic and semantic types

```
HB.mixin Structure isML_universe (ml_type : Type) := {  
  eqclass : Equality.class_of ml_type ;  
  coq_type : forall M : Type -> Type, ml_type -> Type ;  
  ml_nonempty : ml_type ;  
  val_nonempty : forall M, coq_type M ml_nonempty }.
```

- ▶ we use HIERARCHY-BUILDER to combine the syntax and interpretation \implies an “ML_universe”.
- ▶ additional `ml_nonempty` and `val_nonempty` assures the existence of at least one nonempty type

Outline

Overview

COQ semantics of OCAML types

Monadic Semantics of OCAML Programs

Examples

Conclusions

Typed store monad - a global monad for OCAML

The **M** in

```
Fixpoint coq_type63 (M : Type -> Type) (T : ml_type) : Type :=
  match T with
  | ml_int => int
  | ml_bool => bool
  | ml_unit => unit
  | ml_arrow T1 T2 => coq_type63 T1 -> M (coq_type63 T2)
  | ml_ref T1 => loc T1
  | ml_rlist T1 => rlist (coq_type63 T1) T1
  end.
```

needs to handle all OCAML effects

- ▶ mutable values (references)
- ▶ failures
- ▶ exceptions, etc.

We define the “typed store monad” to model the first two.

Defining a monad with MONAE

We rely on MONAE to handle definitions about monads:

- ▶ define the interface (operators and theory) of a monad
to write equational proofs on programs
- ▶ prove and define instances of the interface
to see the consistency and properties of the interface

These definitions are systematically organized using
HIERARCHY-BUILDER.

The typed store monad

- ▶ In the *interface* part, the typed store monad inherits the basic monad interface that has only `bind` and `ret`, adding four operators (`cnew`, `cget`, `cput`, `crun`) and several equations
- ▶ In the *model* part, we give executable definitions of operators and prove the equations for them.

For example, here is the interface and model of `cget`:

```
(in hierarchy.v)
```

```
cget : forall {T}, loc locT T -> M (coq_type M T) ;
```

```
(in typed_store_model.v)
```

```
Let cget T (r : loc T) : M (coq_type T) :=  
  fun st =>  
    if nth_error (ofEnv st) (loc_id r) is Some (mkbinding T' v) then  
      if coerce T v is Some u then inr (u, st) else inl tt  
    else inl tt.
```

- ▶ `coerce` is a boolean function that compares a type `T : ml_type` with the type of some value `v : coq_type M T'`

Dynamic type checking: `coerce`

```
Definition coerce (T1 T2 : X) (v : f T1) : option (f T2) :=  
  if @eqPc _ T1 T2 is ReflectT H then Some (eq_rect _ _ v _ H) else None.
```

```
Definition cget T (r : loc T) : M (coq_type M T) :=  
  fun st =>  
    if nth_error st (loc_id r) is Some (mkbinding T' v) then  
      if coerce T v is Some u then Ret (u, st) else fail  
    else fail.
```

- ▶ `coerce` assures that an access to the store is correctly typed
 \implies the **dynamically** typed store monad
- ▶ dynamic type checking needs dynamic type comparison
 \implies the syntactic types are necessary

Combining things into a model

$$MA := Env \rightarrow (1 + Env \times A)$$

Section `predef`.

Variable `ml_type` : `ML_universe`. *(* has a canonical coq_type *)*

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

Arguments `mkbind` {`M` `bind_type`}.

Definition `MO` `Env` (`T` : `UU0`) := `MS` `Env` `option_monad` `T`.

(transformer MS provides the monad interface *)*

End `predef`.

`#[bypass_check(positivity)]`

Inductive `Env` (`ml_type` : `ML_universe`) :=
 `mkEnv` : `seq` (`binding` `ml_type` (`MO` (`Env` `_`))) -> `Env` `_`.

(entangle the monad and environment *)*

Section `def`.

Variable `ml_type` : `ML_universe`.

Definition `M` (`Env` `ml_type`) (`T` : `UU0`) := `MS` `Env` `option_monad` `T`.

End `def`.

Equations of the Typed Store Monad

Equations are basic reasoning tools that relates the operators of the monad

Sample relation between `cget` and `cnew`:

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

- ▶ Direct paraphrase: the `cnew` operator does not change the meaning of `cget`
- ▶ Intuition: this equation expresses the “freshness” of locations

Not that in practice, we rather use a “derived” equation:

```
Lemma cchknewget T T' (r : loc T) s (A : UU0) k :  
  cchk r >> (cnew T' s >>= fun r' => cget r >>= k r') =  
  cget r >>= (fun u => cnew T' s >>= k ^^ u) :> M A.
```

Outline

Overview

COQ semantics of OCAML types

Monadic Semantics of OCAML Programs

Examples

Conclusions

fibonacci

```
Fixpoint fibo_ref n (a b : loc ml_int) : M unit :=
  if n is n.+1 then
    cget a >>= (fun x => cget b >>= fun y => cput a y >> cput b (x + y))
      >> fibo_ref n a b
  else skip.
```

```
Fixpoint fibo_rec n :=
  if n is m.+1 then
    if m is k.+1 then fibo_rec k + fibo_rec m else 1
  else 1.
```

```
Theorem fibo_ref_ok n :
  crun (cnew ml_int 1 >>=
    (fun a => cnew ml_int 1 >>= fun b => fibo_ref n a b >> cget a))
  = Some (fibo_rec n).
```

factorial on Int63

```
Definition fact_for63 (n : coq_type ml_int) : M (coq_type ml_int) :=
  do v <- cnew ml_int 1%int63;
  do _ <-
    (do u <- Ret 1%int63;
     do v_1 <- Ret n;
     forloop63 u v_1
      (fun i =>
        do v_1 <- (do v_1 <- cget v; Ret (mul v_1 i));
        cput v v_1));
  cget v.
```

```
Theorem fact_for63_ok :
  crun (fact_for63 (N2int n)) = Some (N2int (fact_rec n)).
```

cyclic graph

```
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) T);
do l <-
(do v <- cnew (ml_rlist T) (Cons (coq_type T) T b r);
  Ret (Cons (coq_type T) T a v));
do _ <- cput r l; Ret l.
```

```
Definition hd (T : ml_type) (def : coq_type T)
  (param : coq_type (ml_rlist T)) : coq_type T :=
match param with | Nil => def | Cons a _ => a end.
```

```
Lemma hd_is_true :
  crun
  (do l <- cycle ml_bool true false; Ret (hd ml_bool false l))
  = Some true.
```

Outline

Overview

COQ semantics of OCAML types

Monadic Semantics of OCAML Programs

Examples

Conclusions

Construction of the typed store monad

Monad interface = (type of) operators + equations

A model consists in

- ▶ an implementation of the operators
- ▶ proofs that the equations are valid

In our work, a model has two purposes:

1. validate the equations
2. be executable

Two models:

- ▶ `monad_model.v`: does not require axioms
- ▶ `typed_store_model.v`: requires bypass of positivity check

Technical note: monad transformers from `MONAE` help writing the model of the typed store monad while keeping proofs “readable” (can be displayed in a small screen and principled)

Comparison with the ST monad

The ST monad [Launchbury and Jones, 1994, Sect. 2.2] has similarly typed operations as our typed store monad:

- ▶ ST monad: `runST : forall a, (forall s, ST s a) -> a`
- ▶ Ours: `crun : forall a, M A -> option a`

the universal parameter `s` to `ST` is used to distinguish different levels of `runST`'s; `STRef` is also similar to `loc`:

- ▶ ST monad: `newST : forall a, a -> ST s (STRef s a)`
- ▶ Ours: `cnew : forall a, coq_type a -> M (loc a)`

Uses of HB

- ▶ extend the hierarchy without mistakes (declarations of coercions and canonical instances are error-prone)
- ▶ flexibly combine existing monads and transformers to build models
- ▶ parametrize models by various ML universes, attaching different universe structures onto an `ml_type`

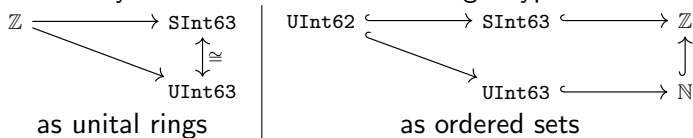
Future work

- ▶ Regarding `ML_universe` as a Tarski universe

$$\frac{\tau : \text{ml_type}}{\text{coq_type } \tau : \text{Type}}$$
 suggests further extension of our approach

by means of induction-recursion [Dybjer and Setzer, 2003], especially to GADTs

- ▶ More library for structures between integer types



- ▶ `let rec f x = ...`

- ▶ `COQGEN` can translate `let rec` with a fuel parameter
- ▶ no equational theory about the fuel in `MONAE` yet



Affeldt, R., Nowak, D., and Saikawa, T. (2019).

A hierarchy of monadic effects for program verification using equational reasoning.

In *MPC 2019*.

<https://github.com/affeldt-aist/monae>.



Cohen, C., Sakaguchi, K., and Tassi, E. (2020).

Hierarchy Builder: Algebraic hierarchies made easy in Coq with Elpi (system description).

In *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020), June 29–July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPICs*, pages 34:1–34:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.



Dybjer, P. and Setzer, A. (2003).

Induction–recursion and initial algebras.

Annals of Pure and Applied Logic, 124(1):1–47.



Garrigue, J. and Saikawa, T. (2022).

Validating OCaml soundness by translation into Coq.

In *TYPES 2022*.



Launchbury, J. and Jones, S. L. P. (1994).

Lazy functional state threads.

In *the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20–24, 1994*, pages 24–35. ACM.