Environment-friendly monadic equational reasoning for OCaml

Reynald Affeldt[1], Jacques Garrigue[2], and Takafumi Saikawa[2]

[1] National Institute of Advanced Industrial Science and Technology (AIST), Japan
[2] Graduate School of Mathematics, Nagoya University, Japan

Abstract

In order to formally verify OCaml programs, we extend a Coq formalization of monadic equational reasoning with a monad to represent typed stores together with its equational theory. We combine this result with the output of CoqGen, an experimental compiler from OCaml to Coq, and demonstrate its usefulness with a few examples.

CoqGen [4] is an extension of the OCaml compiler that translates OCaml programs to Coq to validate OCaml type checking. Monae [2] is a Coq library to verify programs using monadic equational reasoning. In order to verify OCaml programs, we extend Monae with a new monad and its equational theory so that programs generated by CoqGen can be verified in Coq. We call this monad the *typed-store monad* because it consists essentially of a mutable typed store. Using this approach, we can effectively reuse the output of CoqGen, instead of discarding it as a mere witness of type checking, and give it a second life as a target for formal verification.

Representation of OCaml types The types of OCaml are represented by the following module interface. In order to encode various OCaml functionalities, types are translated in two steps: (1) they are represented as syntactic trees (`ml_type`[1]) with decidable equality and (2) they are interpreted into Coq types in the style of a Tarski universe [5]:

```
Parameter ml_type : Set.
Variant loc : ml_type -> Type := mkloc T : nat -> loc T.
Parameter coq_type : forall M : Type -> Type, ml_type -> Type.
```

The `loc` identifier above is for memory locations. The interpretation function `coq_type` [4] is parameterized by the (yet to be defined) typed-store monad. Concrete definitions for `ml_type` and `coq_type` are generated by the CoqGen compiler. The presence of the Tarski universe is the main difference with `coq-of-ocaml` [3] and `hs-to-coq` [6].

The typed-store monad interface We have designed an interface for the typed-store monad of CoqGen and implemented it using Hierarchy-Builder:

```
HB.mixin Record isMonadTypedStore (M : Type -> Type) of Monad M := {
  cnew : forall {T}, coq_type M T -> M (loc T) ;
  cget : forall {T}, loc T -> M (coq_type M T) ;
  cput : forall {T}, loc T -> coq_type M T -> M unit ;
  crun : forall {A : Type}, M A -> option A ;  ... (* laws omitted *) }.
```

It consists of four operations: `cnew` to create locations, `cget` to dereference a location, `cput` to update a location, and `crun` to execute a monadic computation in an empty store. To the best of our understanding, these operations are similar to the ST monad of Haskell. They are sufficient to represent OCaml programs generated by CoqGen. The interface is completed by seventeen equations. Several of them are reminiscent of the state monad. Others involving the `cnew` operation are specific to the typed-store monad, for example:

```
cnewget : forall T (s : coq_type M T) A (k : loc T -> coq_type M T -> M A),
  cnew s >>= (fun r => cget r >>= k r) = cnew s >>= (fun r => k r s) ;
```

See `hierarchy.v` for the complete interface. We are not aware of a description of the equational theory of the ST monad of Haskell. The model of operations comes from CoqGen, and we ported it to Monae using monad transformers (it is obtained by applying the state monad transformer [1] to the failure monad):

[1]For the implementation, see PR #105, branch `typed_store_monad` in Monae [2].

```
Record binding (M : Type -> Type) :=
  mkbind { bind_type : ml_type; bind_val : coq_type M bind_type }.
Definition M0 Env (A : Type) := MS Env option_monad A.
(* we locally disable strict-positivity checking to allow functions in stores² *)
#[bypass_check(positivity)]
Inductive Env := mkEnv : seq (binding (M0 Env)) -> Env.
Definition M := M0 Env. (* action on objects *)
```

See typed_store_model.v for the proofs of interface equations. Note that even though the interface does not feature any use of the fail operation (from the failure monad) we use the option monad (an instance of the failure monad) to build the model of the interface. It is required to model memory access errors (including type errors).

Program verification using the typed-store monad The basic idea to verify programs by monadic equational reasoning is to use the interface of the typed-store monad. Library lemmas also need to be derived from the interface. In practice we observe that there is a need for at least one more operation to "check" the validity of a memory location (similarly to assertions in Hoare logic) so as to produce commutation lemmas. It is derived from the interface:

```
Definition cchk T (r : loc T) : M unit := cget r >> skip.
```

It allows to formalize commutation lemmas such as the following one:

```
Lemma cchknewget T T' (r : loc T) s (A : Type) k :
  cchk r >> (cnew T' s >>= fun r' => cget r >>= k r') =
  cget r >>= (fun u => cnew T' s >>= k ^~ u) :> M A.
Proof. by rewrite bindA bindskipf cputnewC. Qed.
```

The ability to express such derived lemmas was an important guiding principle to design the interface equations.

Examples The file example_typed_store.v provides a number of verification examples. Let us consider the verification in Coq of an imperative implementation of factorial in OCaml:

```
let fact_for63 n =
  let v = ref 1 in for i = 1 to n do v := !v * i done; !v;;
```

Generating the corresponding Coq formalization is as easy as running `ocamlc -c -coq fact.ml`[3]:

```
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.
```

As for the correctness statement, we use the fact_rec implementation from MathComp:

```
Hypothesis Hn : (Z.of_nat n < Sint63.to_Z Sint63.max_int)%Z.
Theorem fact_for63_ok : crun (fact_for63 (N2int n)) = Some (N2int (fact_rec n)).
```

The function N2int turns natural numbers into 63 bit integers. We have verified this theorem using only monadic equational reasoning in a matter for 28 lines of script excluding library lemmas.

Future work We have not yet formalized in Coq the exceptions of the typed-store monad of CoqGen; we plan to do so with monad transformers. A more open problem is the question of how to handle fuel, which is used in CoqGen to allow unrestricted recursion.

[2]This allows the definition of non-terminating functions in Coq, which leads to an inconsistency. However, Monae programs are not executable in Coq since their models are hidden by an interface.

[3]For the implementation, see the Coqgen PR [4].

References

[1] R. Affeldt, D. Nowak. Extending equational monadic reasoning with monad transformers. In *TYPES 2020*.

[2] R. Affeldt, D. Nowak, T. Saikawa. A hierarchy of monadic effects for program verification using equational reasoning. In *MPC 2019*. `https://github.com/affeldt-aist/monae`.

[3] G. Claret. *Program in Coq. (Programmer en Coq)*. PhD thesis, Paris Diderot University, France, 2018.

[4] J. Garrigue, T. Saikawa. Validating OCaml soundness by translation into Coq. In *TYPES 2022*.

[5] P. Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in proof theory*. Bibliopolis, 1984.

[6] A. Spector-Zabusky, J. Breitner, C. Rizkallah, S. Weirich. Total Haskell is reasonable Coq. In *ICFP 2018*.