Validating OCaml soundness by translation into Coq

Jacques Garrigue and Takafumi Saikawa

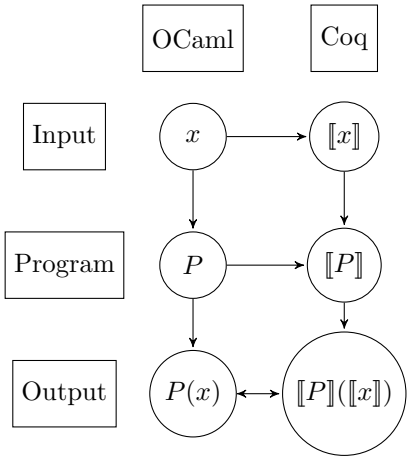
Graduate School of Mathematics, Nagoya University

The programming language OCaml is known for the expressiveness of its type system. Starting with an ML core consisting of algebraic datatypes, mutable references, and exceptions, it extends it with polymorphic variants and objects, extensible datatypes, GADTs, a module system, and even first class modules. Further extensions are planned.

Proving the correctness of both theory and implementation for such a complex system is a daunting task. Formalization of the theory of ML languages started in the 1990's, both in HOL [13], and Coq [1], and culminated with the formalization of full SML in Twelf [7]. However, this is only with Cake ML that the proof included the actual implementation [6, 12], albeit for a much smaller language. Concerning OCaml, most parts of the type system were proved correct on paper, but only independently of each other. Some parts were further formalized in Coq [4, 8], but independently of the actual implementation, which often differs in subtle ways. A formal semantics for the core of the language has also been given, but again independently of the implementation [9].

An alternative to actually proving things is to check them a posteriori. That is, rather than proving the OCaml typechecker correct, verify that its output is correct. That is the approach taken by Couderc [3], by writing a type checker to be applied to the so called *typed tree*, which is the result of type inference. While redundancy provides some extra guarantees against simple implementation bugs, this approach may still leave some bugs, and does not ensure that the type system itself is sound.

We propose here a new approach, based on the translation of OCaml programs to Gallina. By implementing it as an extra backend to the OCaml compiler, taking the *typed tree* as input, we can use Coq's type checker to ensure the type soundness of source programs, as long as we can assume that the semantics is unchanged. The type soundness is a consequence of Coq's subject reduction property, following the principle shown in the following figure.



Namely, if for all $P : \tau \to \tau'$ and $x : \tau$ we have:

1. P translates to $[\![P]\!]$, and $\vdash [\![P]\!] : [\![\tau \to \tau']\!]$
2. x translates to $[\![x]\!]$, and $\vdash [\![x]\!] : [\![\tau]\!]$
3. $[\![P]\!]$ applied to $[\![x]\!]$ evaluates to $[\![P(x)]\!]$
4. $[\![\cdot]\!]$ is injective on types

then the soundness of Coq's type system implies the soundness of OCaml's evaluation (i.e., evaluation of well-typed programs cannot go wrong).

This approach has a relatively large trusted base: we assume that the translator does not change the semantics of the input program, when using Coq normalization, and that Coq indeed enjoys type soundness [10].

The goal of relying on Coq for the soundness has led us to a number of design choices. Since we want to be able to evaluate programs inside Coq, we realize side effects through a concrete monadic implementation, and avoid using axioms that would block evaluation. We

also avoid unsound extensions of Coq as much as possible. Finally, as references (and polymorphic comparison) require some form of dynamic type information, we provide an intensional representation of types.

Extraction from Coq to other languages has been an active research area since its beginning, but the opposite direction is much more recent. Currently, two other translators are available: `coq-of-ocaml` [2] and `hs-to-coq` [11], for OCaml and Haskell respectively. However, their goal is different from ours, as they intend to prove properties on the translated programs, and as a result are happy to restrict themselves to a subset of the language in order to get simpler translated code. In particular they do not introduce an intensional representation of types.

The current version of the translator [5] only supports the core part of the language: Hindley-Milner polymorphism, algebraic datatypes, polymorphic comparison, references and exceptions. However, we already have extensions in mind, and in particular the choice of having an intensional representation of types should also be an advantage when translating GADTs, as they rely on the ability to compare types, which in Coq is only possible if they have a matchable representation. Due to our need to aggregate all defined types, the translator is restricted to single-file programs, but this restriction should also be eventually lifted, by adding a linking phase.

Since we intend to translate faithfully arbitrary OCaml programs to Coq, we need to recreate the OCaml world inside Coq. The two essential parts of it are:

- defining a monad $M\ T = Env \to Env \times (T + Exn)$ for state (references), exceptions, and non-termination (seen as a non-catchable exception),
- defining a translation from the intensional representation of types to their Coq semantics.

Here we will just show the essential part of the functor `REFmonad` that creates such a monad from the intensional representation of types `ml_type` and its translation `coq_type`.

```
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 *)
  ... (* Monadic operations for references and exceptions *)
End REFmonad.
```

You can see that `coq_type` itself depends on the monad, as it is needed when translating function types. As a result, the mutually recursive definition of `Env` and `Exn` clearly violates the positivity condition enforced by Coq. This comes as no surprise, since it is well known that references can be used in ML to define non-terminating functions, in the absence of recursion. However, we conjecture that disabling the positivity check for this definition alone does not endanger Coq soundness, as non-termination is restricted to computations happening inside the monad; namely, we can indeed create values of type `M False`, but we cannot extract `False` from the resulting sum type.

References

[1] Olivier Boite and Catherine Dubois. Proving type soundness of a simply typed ML-like language with references. In *Proc. of the International Conference on Theorem Proving in Higher Order Logics*, 2001.

[2] Guillaume Claret. Coq of OCaml. In *OCaml Users and Developers Meeting*, August 2014.

[3] Pierrick Couderc. *Vérification des résultats de l'inférence de types du langage OCaml*. PhD thesis, Université Paris-Saclay, October 2018.

[4] Jacques Garrigue. A certified implementation of ML with structural polymorphism and recursive types. *Mathematical Structures in Computer Science*, 25(4):867–891, November 2014.

[5] Jacques Garrigue. OCaml in Coq. GitHub PR, 2022. https://github.com/COCTI/ocaml/pull/3.

[6] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Proc. 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 179–191, New York, NY, USA, 2014. Association for Computing Machinery.

[7] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. In *Proc. 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, page 173–184, New York, NY, USA, 2007. Association for Computing Machinery.

[8] Thomas Leventis. A row Coq proof of soundness for the relaxed value restriction. https://www.irif.fr/~leventis/pub/Stages/Coq/, September 2012.

[9] Scott Owens. A sound semantics for OCaml light. In *Proc. European Symposium on Programming*, volume 4960 of *Springer LNCS*, pages 1–15, April 2008.

[10] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq Correct! verification of type checking and erasure for Coq, in Coq. *Proc. ACM Program. Lang.*, 4(POPL), January 2020.

[11] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. Total Haskell is reasonable Coq. In *Proc. International Conference on Functional Programming*, pages 14—-27, New York, NY, USA, 2018.

[12] Yong Kiam Tan, Scott Owens, and Ramana Kumar. A verified type system for CakeML. In *Proc. 27th Symposium on the Implementation and Application of Functional Programming Languages*, IFL '15, New York, NY, USA, 2015. Association for Computing Machinery.

[13] Myra Ellen VanInwegen. *The machine-assisted proof of programming language properties*. PhD thesis, University of Pennsylvania, August 1996.