Coq of OCaml

OCaml Workshop 2014

1 Introduction

The CoqOfOCaml project is a compiler from a subset of the OCaml language to the Coq programming language. This compiler aims to allow reasoning about OCaml programs, or to import existing OCaml libraries in Coq.

Chain of compilation The CoqOfOCaml compiler imports an OCaml program foo.ml into a Coq file foo.v. We encode effects using a monadic translation [2], since Coq allows only purely functional and terminating functions, mimicking the way the Haskell language handles imperative traits. These effects can be run within Coq, although very slowly. Reversely, we can compile foo.v into an OCaml file foo2.ml. This compilation removes all the proof terms and translates encoded effects to concrete imperative OCaml effects.

```
    foo.ml      CoqOfOCaml
    foo.v      compilation
    foo2.ml
```

OCaml   Coq with monads   OCaml

Specification The specification of our tool is twofolds. First, we require that foo2.ml behaves as foo.ml (has the same input-output traces). Second, we require that foo2.ml behaves as foo.v when we interpret the effects in Coq. Therefore the user can continue to work on and update foo.v once the original program is imported. This is the case of the development process in which the user imports foo.ml just once at the beginning, and only works on foo.v afterward.

Effects system Until now, we focused on the import process, not on the proof techniques on Coq programs. The import is based on a type and effects system [3]. We compose an effects inference with a monadic translation guided by the effects. We chose not to encode effects in one big generic monad but in many fine grained and composable ones. We hope it leads to simpler specifications and proofs, localizing effects only where they are needed. We rely on the dependent types to represent these composable effects in Coq.

Related work Other tools have been developed to formally verify functional programs with effects in Coq, like CFML, Ynot or Why. Our contribution is novel because we generate a shallow embedding in Coq of an existing higher-order language (OCaml), with the ability to extract it back to an equivalent program with effects.

We will present our effects system, explain how our implementation works, show some case studies and conclude with future directions.

2 Design

We will describe the effects system, how effects are inferred in OCaml and how effectful terms are represented in Coq.

Effects For this first version, we restricted ourselves to a simple effects system with no polymorphic effects, keeping that part as a future work.

An effects descriptor is an unordered set of atomic descriptors, acting as names for exceptions, top-level references or the special input-output descriptor:

\[
    d := \{a_1, \ldots, a_n\}
\]

An effect type is the shape of an OCaml type with additional effects information. The syntax is the following:

\[
    \begin{align*}
    \tau & ::= \text{Pure} \\
    & \mid \text{Pure } d \to \tau
    \end{align*}
\]

So an expression can be pure (as effect-free and terminating) or a function generating the descriptor \(d\) when applied. In particular arguments of a function have to be effects-free.

Types and effects inference in OCaml We first infer the datatypes using the OCaml compiler’s front-end. Then we derive the effects of each sub-expression in a bottom-up analysis with an environment of effects. For mutually recursive functions, we compute a fixpoint until convergence.

Representation in Coq We use a monadic encoding to represent effects in Coq. Given a type \(A\) and monad \(M\), an expression \(e\) of type \(M A\) is a computation returning a value of type \(A\).

To compose computations there are two basic operators:

\[
    \begin{align*}
    \text{return} : & \forall A, A \to M A \\
    \text{bind} : & \forall A B, M A \to (A \to M B) \to M B
    \end{align*}
\]

We associate a monad \(M_d\) to each descriptor \(d\), definable in Coq thanks to dependent types. For a \(d\) given by \(\{a_1, \ldots, a_n\}\):

\[
    M_d A = (S_1 \times \cdots \times S_n) \to (A+E_1+\cdots+E_n) \times (S_1 \times \cdots \times S_n)
\]

where \(S_i\) is the mutable state of \(a_i\) and \(E_i\) the error which can be raised by \(a_i\). We chose an arbitrary order to index the atomic descriptors \(a_i\), knowing that there is a canonical isomorphism between two definitions given two different orders. This monad is composable: \(M_{d_1}\) and \(M_{d_2}\) compose into \(M_{d_1 \cup d_2}\). Besides, this composition is commutative, in opposition to the composition of monad transformers.

To combine different computations with different descriptors we use the lift operator. If \(d \subset d'\):

\[
    \text{lift}_{d, d'} : \forall A, M_d A \to M_{d'} A
\]
Two different computations of descriptors $d_1$ and $d_2$ can be lifted to two computations of descriptor $d_1 \cup d_2$ and composed with the standard bind operator of $M_{d_1 d_2}$.

The monads for a reference of type $S$ and an exception carrying a value of type $E$ (with $\emptyset$ the empty type) are defined as:

\[
\begin{align*}
M_{\text{Ref}} A & = S \to (A + \emptyset) \times S \\
M_{\text{Exn}} A & = \text{unit} \to (A + E) \times \text{unit}
\end{align*}
\]

On the contrary, effects like the non-termination are encoded using references and exceptions. This effect is used to represent non-terminating functions or functions whose termination is not proven. The non-termination is the composition of a reference to a decreasing counter (the fuel) and of an exception raised on non-terminated computations (when the counter reaches zero).

3 Implementation

The CoqOCaml compiler is implemented in OCaml. It imports a subset of the OCaml abstract syntax tree typed by the OCaml compiler front-end. Then it infers effects, does the monadic translation and pretty-print the output in the Coq syntax.

We support the pure lambda-calculus kernel, mutually recursive definitions, inductive types definitions with pattern-matching, records, abstract types and modules. We do not handle signatures and functors. We have exceptions and global references. We support the main parts of the Pervasives and List libraries.

This compiler was challenging since we needed to import a large subset of OCaml to start working on real programs. We decided not to support functors due to code complexity. We may be able to reduce this code complexity using the new annotation mechanism of OCaml to directly annotate nodes of the syntax tree with effects.

A lot of care was given to get an output both readable and close to what a real programmer could write. Indeed, the user is supposed to work on the Coq version once the OCaml code had been imported.

4 Case studies

We successfully imported the slightly modified List, Set and Map modules from the standard OCaml library. These modules work on immutable structures but are not purely functional: they contain exceptions and functions whose termination is not obvious.

Here is an example of the map2 function as defined in the OCaml library:

```coq
let rec map2 f 11 12 =
  match (11, 12) with
  | (1, 1) -> []
  | (a1, a2; : : 12) ->
    let r = f a1 a2 in r :: map2 f l1 l2
  | (_, _) -> invalid_arg "List.map2"

Fixpoint map2 {A B C : Type} (f : A -> B -> C) (l1 : list A) (l2 : list B) :=
  match (l1, l2) with
  | (l1, l2) => return []
  | (cons a1 l1, cons a2 l2) =>
```

The size increase was mainly due to the monadic translation, especially in the case of non-termination where we define an auxiliary function. We hope this size increase is small enough so the user can continue to work easily on the imported Coq files.

We have not worked on the reasoning rules about monadic programs yet. Still, it was possible to manually make a proof in Coq about an absence of exception in the generated code: in the List module, the sort function depends on an auxiliary function which may raise an exception Assert_failure. Its importation into Coq is:

```coq
Fixpoint chop {A : Type} (k : Z) (l : list A) :
  \ exists \ struct l \ : \ 0 <= k \ <= \ length l \ -> \ { l' : list A | length l' = length l - k }.

Using this new chop function, we updated the definition of sort and proved that it never raises an exception either.

5 Conclusion

We have presented a tool, CoqOCaml, which can import existing examples of OCaml programs to Coq and extract them to OCaml. An effects system and a monadic translation is used to represent effectful programs in Coq. The sources can be downloaded on https://github.com/clarus/coq-of-ocaml.

In the future we would like to investigate more the programming and proof techniques on Coq programs with effects. Some interesting problems are the extension of the effects system to handle the polymorphism, an implementation of an effects inference mechanism on Coq terms, the representation of new effects (including concurrency), the design of reasoning rules on monadic programs with dependent types, and the certification of an extraction chain to OCaml with effects.

References