Synthesis of certified programs with effects using monads in Coq

Sara Fabbro and Marino Miculan

Laboratory of Models and Applications of Distributed Systems
Department of Mathematics and Computer Science, University of Udine, Italy
fabbro.sara.88@gmail.com, marino.miculan@uniud.it

An important feature of type-theory based proof assistants is the possibility of extracting *certified* programs from proofs [4, 2], in virtue of the Curry-Howard "proofs-as-programs", "propositions-as-types" isomorphism. The extracted programs are certified in the sense that they are guaranteed to satisfy their *specification*, i.e. the properties represented by their types in the proof assistant.

One limitation of this approach is that these programs are always purely functional. Non-functional programming languages (e.g. imperative, distributed, concurrent,...) hardly feature a type theory supporting a Curry-Howard isomorphism, and even if such a theory were available, implementing a specific proof-assistant with its own extraction facilities would be a daunting task.

In this talk we present a methodology for circumventing this problem using the extraction mechanisms of existing proof assistants (namely Coq), generalizing previous work [3]. Basically, the idea is to incapsulate the non-functional aspects in a *computational monad*, as done in Haskell, and using the extraction facilities of Coq for directly producing certified Haskell code with monads.

Let us consider an algebraic specification (T, Σ, Γ) of a monad T. This consists of an abstract type constructor T (i.e., for each type A, TA is the type of computations whose values have type A), and a set $\Sigma = \{op_1, \ldots, op_n\}$ of (multi-sorted) constructors for the monadic types TA. The behaviour of these constructors is specified by the set $\Gamma = \{s_1 = t_1, \ldots, s_m = t_m\}$ of equational laws; terms s_i, t_i in these equations are built using the operators in Σ, plus the basic constructors of any monad $return_A : A \to TA$ and $bind_{A,B} : TA \to (A \to TB) \to TB$ (often written $\gg=$). For instance, the "maybe" monad M is defined by a single (polymorphic) constructor $nothing_A : MA$, and a single equation $bind_{A,B}(nothing_A, f) = nothing_B$. Similarly, the "global store" monad can be specified by two operations *lookup* and *update*, and seven equational laws [7]. Many other computational aspects can be specified in this way; see e.g. [6].

The specification (T, Σ, Γ) is encoded in Coq as a module signature, i.e., `Module Type` specializing `MONAD_INTERFACE`, like the following:

```
Module Type MAYBEMONAD_INTERFACE <: MONAD_INTERFACE.
Parameter Nothing : forall (A: Type), M A.
Axiom Strictness : forall (A B : Type) (f : A -> M B),
   (Nothing A) >>= f  =  (Nothing B).
End MAYBEMONAD_INTERFACE.
```

Then, we can start reasoning about (and implementing) programs with effects by *assuming* a monad implementing this signature. Program specifications can be given using the equational logic at the `Prop` level of Coq. For instance, the specification of a program for in-place swapping of two locations, in the monad for global store, is the following:

```
Module StateInstance <: STATEMONAD_INTERFACE.
Include STATEMONAD_INTERFACE.
Include MemoryState.

Lemma swap_locs : forall (l1 l2 : loc), l1 <> l2 -> {c : M unit |
 ((c >>= (fun _ => lookUp l2)) =
             (lookUp l1) >>= fun x => c >>= (fun _ => ret x)) /\
 ((c >>= (fun _ => lookUp l1)) =
             (lookUp l2) >>= fun x => c >>= (fun _ => ret x)) /\
 forall (l : loc), (l <> l1 /\ l <> l2) -> ((c >>= fun _ => lookUp(l))) =
              ((lookUp(l) >>= fun x => c >>= fun _ => ret x ))}.
```

This kind of `Lemma`ta can be proved constructively as usual, by providing a program `c` and proving that it meets the specification. This proof will make use of the abstract algebraic laws declared in the monad signature (`STATEMONAD_INTERFACE` in this case). Notice that there is no need to provide any real implementation of the monadic specification in Coq, in order to program with the operators and prove the specification. Actually, it is not even advisable: for proving that programs are compliant to their specifications we cannot rely on peculiar properties of any specific implementation.

At this point, from these proofs we can extract Haskell programs by taking advantage of the standard Coq `Extraction` facility. The programs so obtained cannot be executed, because they will contain the constructors op_i which have still to be defined. Differently from previous work [3], we solve this issue by automatically replacing during the `Extraction` each op_i with a suitable Haskell code fragment, possibly using operators of the corresponding Haskell monad. In the case of the "maybe" monad above, we declare:

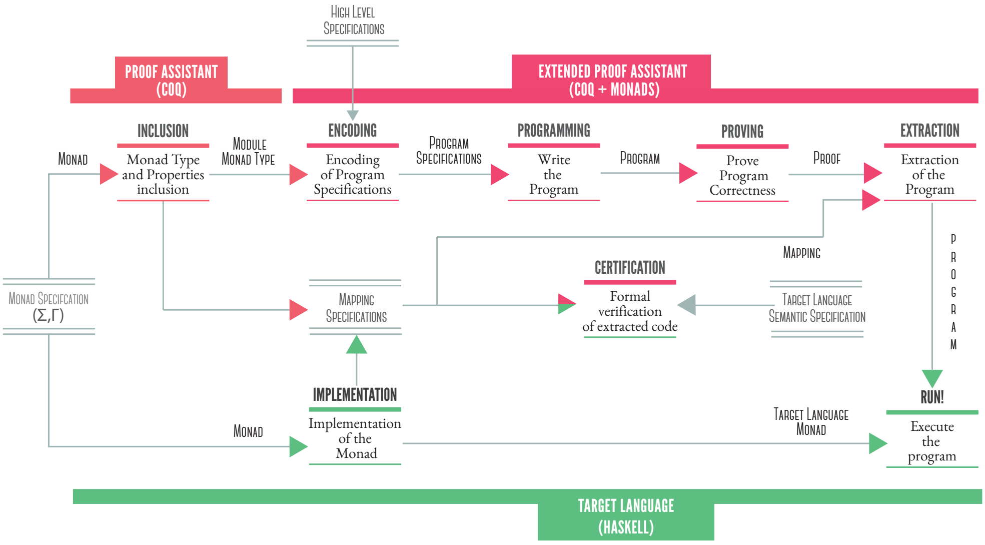
```
Extract Constant Maybe.M "a" => "Maybe a".
Extract Constant Maybe.ret => "Just".
Extract Constant Maybe.bind => "(>>=)".
Extract Constant Maybe.Nothing => "Nothing".
```

This is the step where we provide the implementation of the monad; in general, each operator can be mapped to an arbitrary complex code snippet. With these definitions, the extracted code can be readily executed in the Haskell runtime, with the proper monads covering the non-functional computational aspects.

Still, we have to prove that the mappings defined in the `Extract Constant` declarations are sound. This corresponds to prove that the equational laws declared in the monad interface are respected. The methodology for proving this soundness is general and uniform, and proceeds as follows. Let $s = t$ be an equational law of the monad specification, and let s', t' the two Haskell programs obtained by extraction from s, t, respectively; we have to prove that s' and t' are *semantically equivalent* with respect to the semantics of Haskell.

Now, instead of working with the full-blown Haskell syntax and semantics, it is more convenient to work with the *Core language*, a very small, explicitly-typed, variant of System F used as an intermediate language in `ghc`. On this language we can easily define an *applicative bisimulation* $M \approx N$ à la Abramsky [1], which corresponds to the behavioural (i.e., contextual) equivalence. Thus, for each pair s', t' as above, let s'', t'' be the corresponding two Core terms produced by `ghc` (with suitable options); we have to prove that $s'' \approx t''$. Once all equivalences $s'' \approx t''$ have been proved, we can assert that the mapping defined by the `Extract Constant` clauses is correct with respect to the equational laws assumed in the monad signature, and hence the extracted code with effects is certified.

These equivalences can be proved "on the paper", following the usual techniques for applicative bisimulation. However, these proofs can be quite long and error-prone, hence it is better (and safer) to develop them within a proof assistant. To support these proofs we are currently developing a formalization in Coq of the syntax, semantics and behavioural equivalence of Core language, similar to that in [5]. The whole methodology is summarized in the following diagram.


High Level Specifications
Proof Assistant (Coq)
Extended Proof Assistant (Coq + Monads)
Inclusion
Monad Type and Properties inclusion
Monad
Module Monad Type
Encoding
Encoding of Program Specifications
Program Specifications
Programming
Write the Program
Program
Proving
Prove Program Correctness
Proof
Extraction
Extraction of the Program
Mapping
Program
Monad Specifcation (Σ,Γ)
Mapping Specifications
Certification
Formal verification of extracted code
Target Language Semantic Specification
Implementation
Implementation of the Monad
Monad
Target Language Monad
Run!
Execute the program
Target Language (Haskell)

References

[1] S. Abramsky. The lazy lambda calculus. *Research topics in functional programming*, pages 65–116, 1990.

[2] P. Letouzey. Extraction in Coq: An overview. In *Proc. CiE*, volume 5028 of *Lecture Notes in Computer Science*, pages 359–369. Springer, 2008.

[3] M. Miculan and M. Paviotti. Synthesis of distributed mobile programs using monadic types in Coq. In *Proc. ITP'12*, LNCS 7406. Springer, 2012.

[4] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[5] M. Pirog and D. Biernacki. A systematic derivation of the STG machine verified in Coq. In *ACM Sigplan Notices*, volume 45, pages 25–36. ACM, 2010.

[6] G. D. Plotkin and A. J. Power. Computational effects and operations: An overview. *Electr. Notes Theor. Comput. Sci.*, 73:149–163, 2004.

[7] G. D. Plotkin and J. Power. Notions of computation determine monads. In *Proc. FoSSaCS*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.