Extraction of certified programs with effects from proofs with monadic types in Coq

Marino Miculan[1⋆] and Marco Paviotti[2⋆⋆]

[1] Dept. of Mathematics and Computer Science, University of Udine, Italy
[2] IT University of Copenhagen, Denmark
marino.miculan@uniud.it, mpav@itu.dk

Abstract. We present a methodology for the extraction of certified programs with effects, using the Coq proof assistant and monadic types.
First, we define in the Calculus of Inductive Constructions an abstract *global store monad*, by suitably adapting Plotkin-Power's sound and complete axiomatization. This monad gives raise to monadic types which can be used for the specifications of imperative programs in Coq. From the (constructive) proofs of these specifications, using Coq's `Extraction` facility we can extract programs in a *computational metalanguage CompML*, which is basically the functional fragment of ML extended with the monad's operators.
In order to obtain executable code, we define a *back-end compiler* from CompML to the target language, i.e. ML in our case. A program in CompML is translated to executable ML code by replacing monadic constructors with the corresponding SML operations. The back-end compiler is proved to be correct, by formally proving that it respects the axiomatization of the global store monad. Hence, the code it produces is certified.
This methodology can be ported to other computational aspects, by suitably adapting the monadic type theory and the back-end compiler, as long as the target language can be given a formal semantics in Coq.

1 Introduction

An important feature of type-theory based proof assistants, like Coq, is the possibility of extracting *certified* programs from proofs [6,3], in virtue of the well-known 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 (expressed in, e.g., the Calculus of Inductive Constructions).

One limitation of this approach is that these programs are always purely functional. In fact, 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.

⋆ Supported by MIUR PRIN 2010LHT4KM project *CINA*.
⋆⋆ Supported by project *DemTech* funded by Danish Council for Strategic Research.

In the previous work [4], we have presented a methodology for circumventing this problem using the extraction mechanisms of existing proof assistants (namely Coq). Basically, the idea is to extend the proof assistant's type theory (e.g., the Calculus of Inductive Constructions) with a suitable *computational monad* `T : Set -> Set`, covering the specific non-functional computational aspect, similarly to what is done in pure functional languages (e.g., Haskell). These monadic types can be used in the statements of `Propositions` to specify the types of programs, like e.g., `f:A -> (T B)`. These propositions can be proved constructively as usual (possibly using the specific properties of the monad operators), and from these proofs we can extract programs by taking advantage of the standard Coq extraction facility. These programs are in a functional language (e.g. Haskell or ML), extended with the *abstract* constructors of the computational monad representing the non-functional features[3]. Since this functional-monadic language with non implemented operators is not runnable and it has no defined semantic either, we *translate* it to any given language where these non-functional features are implemented: this translation generates a program in the target language by replacing the monad constructors with suitable non-functional code. Thus, the functional-monadic language can be seen as an intermediate language, and the post-extraction translation is a *back-end compiler*.

In [4] we have applied this metodology for synthesizing distributed, mobile programs in Erlang [1], using Haskell as intermediate language. A question which was left open in that work was about the *correctness* of the back-end compiler. This is an important issue in our methodology: the resulting target code is really certified (i.e., we can ensure that it satisfies the specification) only if we prove that the back-end compiler "preserves the intended semantics".

This is the question that we face in this paper: how the back-end compiler can be certified, in order to have a correct extraction mechanism?

To this end, we have to address several issues: how to specify the "intended semantics", define a notion of "semantic preservation", and prove that the compiler is correct according to this notion. And this proof should be certified itself.

In this paper, we address these issues. The first one is how to specify the behaviour of the monad operators. One possibility is to give them explicit definitions, that is, define in Coq all data structures and functions needed for implementing the given computational aspect. This can be seen as giving a "reference implementation" of the programming language. However, this approach would give the user (i.e. the programmer) too much information, as it may allow to prove properties about programs which do not necessarily hold in the target language. The reason is that the internal implementation may rely on peculiar choices, which may be different from the actual implementation in the target language. As an example, an implementation of a store using natural numbers as locations would implicitly allow for "pointer arithmetics", whereas the target language may not have such feature.

[3] Categorically speaking, this is the internal language of the Kleisli category of the computational monad, lifted from the base category.

In order to avoid this problem, we propose that the behaviour of the monad should be specified only by a set of assumptions. These are the only properties that a user (programmer) can have access for proving properties (and extracting programs). Technically, this corresponds to define a `Module Type`, where data structures and monad constructors are declared as `Parameter`s and their behaviour is given by a set of `Axiom`s.

Of course, one may (and should) be concerned about the consistency of an arbitrary set of axioms. This can be solved by providing an internal implementation of the monad, which satisfies the assumptions. Technically, this corresponds to define a `Module` whose type is the `Module Type` defined above. The only purpose of this implementation is to ensure the consistency of these assumptions, and it is *not* provided to the user; in this way, the specific implementation choices cannot be used within Coq for proving properties.

The next step is to formalize and certify the back-end compiler. The idea here is to take the assumptions in the monad type, as a specification for the actual implementation, that is, they must be preserved by the translation. This means that in Coq we have to:

1. formalize the syntax of the intermediate language covered by the `Extraction` facility. This is always the functional fragment of the language used for extraction (e.g., Haskell or ML) extended with the constructors of the monad. For this reason we call this language *Computational ML* (CompML), akin to Moggi's *computational λ-calculus* [5].
2. formalize the target language with its semantics;
3. formalize the back-end compiler as a function between these two languages;
4. define a notion of equivalence over the intermediate language, by "lifting" a semantic equivalence from the semantics of the target language;
5. prove that all the properties assumed in the monad specification are satisfied by the translation. This correspond to prove that the compiler induces an implementation of the monad which satisfies the specification.

In order to exemplify this methodology, in the rest of this paper we will apply it to the extraction of programs with side effects in the imperative fragment of ML. We have chosen this computational aspect because it has been given a sound and complete axiomatization by Plotkin and Power [7], and the target language is close to the intermediate language. (On the other hand, the distribuite, concurrent fragment of Erlang has no formalized semantics yet.)

In Section 2 we define the "global store" monad `GS` in Coq. As mentioned before, this corresponds to two phases: first, we define the `Monad Type` of a global store monad, with abstract notions of stores, locations, and their operations (lookup and update); their behaviour is specified by Plotkin and Power's equational laws (suitably adapted to our situation). Then we provide an internal implementation of this `Module Type`, by instantiating all parameters and operators. Thus, we can safely assume these laws as axioms.

In Section 3 we provide an example of program extraction: we give the specification of a program which swaps the content of two locations without altering the rest of the store, and extract a program in CompML.

The back-end compiler is the subject of Section 4. First we formalize both the target language and the intermediate language, i.e., CompML. In order to deal with variable binders in the syntax, we follow the *locally nameless* approach [2], in which bound variables are represented using de Bruijn indices, whereas free variables are represented using names. For the formalization of the target language, we adopt Charguéraud's implementation of Mini-ML [2], which provides the syntax, the semantics and main results for a fragment of ML with data types, recursion, references and exceptions. Then, we define the compiler from CompML to Mini-ML, and prove its correctness by showing that all equational laws are satisfied by the translation. This result can be seen also as a proof that Charguéraud's implementation is sound with respect to the global store semantics.

Concluding remarks, a summary of the whole methodology, and directions for further work are in Section 5. The Coq code of the whole development is available at `http://sole.dimi.uniud.it/~marino.miculan/CoqCode/GSMonad/`.

2 Monadic types for stateful computations in Coq

In this section we describe the monadic type theory we use for reasoning about stateful computation in Coq. First, we give it as an abstract specification, i.e., all datatypes are declared as `Parameter`s and their behavioral laws as `Axiom`s. Then, we show that this specification is consistent, by giving an internal implementation in Coq. Technically, the specification and the implementation are represented as a `Module Type` and a `Module`, respectively.

2.1 Global store monad: specification

In order to model stateful computations we need to define what a store is, and which operations it supports. Following Plotkin and Power [7], these operations can be taken as abstract operations over abstract data types of locations, values and stores, together with the equational laws defining their behaviour. Therefore, we give the specification of a monad `GS`, with the usual `bind` and `return` plus the two `update` and `lookup` operators.

```
Module Type GSMonad.

Parameter Loc : Set.
Parameter Value : Set.
Parameter Store : Set.

Definition GS (A: Set)  := Store -> A * Store.

Parameter ret : forall (A: Set) (x : A), GS A.
Parameter bind : forall (A B :Set) (a : GS A) (f : A -> GS B), GS B.
Parameter update: forall (l: Loc) (n: Value), GS unit.
Parameter lookup: forall (ref: Loc), GS Value.
...
```

Laws for the monad unit and multiplication

1. $\forall A : Set, \forall t : GS(A), (t \texttt{ >>= ret}) = t.$
2. $\forall A, B : Set, \forall f : A \to GS(B), \forall v : A, (\texttt{ret } v) \texttt{ >>= } f) = (fv).$
3. $\forall A, B, C : Set, \forall f : A \to GS(B), \forall g : B \to GS(C), \forall x : GS(A),$
   $((x \texttt{ >>= } f) \texttt{ >>= } g) = (x \texttt{ >>= } (\lambda y.(fy) \texttt{ >>= } g)).$

Laws for the lookup and update operations

4. $\forall A : Set, \forall l : Loc, \forall x : GS(A), (\texttt{lookup}(l) \texttt{ >>= } (\lambda v.(\texttt{update}(l, v) \texttt{ >>= } x))) = x.$
5. $\forall A : Set, \forall l : Loc, \forall f : Value \to Value \to GS(A),$
   $(\texttt{lookup}(l) \texttt{ >>= } (\lambda x.\texttt{lookup}(l) \texttt{ >>= } (\lambda y.(fxy)))) = (\texttt{lookup}(l) \texttt{ >>= } (\lambda x.(fxx))).$
6. $\forall A : Set, \forall l : Loc, \forall v, v' : Value, \forall x : \texttt{unit} \to GS(A),$
   $(\texttt{update}(l, v) \texttt{ >>= } (\lambda\_.(\texttt{update}(l, v') \texttt{ >>= } x))) = (\texttt{update}(l, v') \texttt{ >>= } x).$
7. $\forall A : Set, \forall l : Loc, \forall v, v' : Value, \forall f : Value \to GS(A),$
   $(\texttt{update}(l, v) \texttt{ >>= } \lambda\_.(\texttt{lookup}(l) \texttt{ >>= } f)) = (\texttt{update}(l, v) \texttt{ >>= } \lambda\_.fv).$
8. $\forall A : Set, \forall l, l' : Loc, \forall f : Value \to Value \to GS(A), l \neq l' \to$
   $(\texttt{lookup}(l) \texttt{ >>= } (\lambda v.(\texttt{lookup}(l') \texttt{ >>= } (\lambda v'.(fvv'))))) =$
   $(\texttt{lookup}(l') \texttt{ >>= } (\lambda v'.(\texttt{lookup}(l) \texttt{ >>= } \lambda v.(fvv')))).$
9. $\forall A : Set, \forall l, l' : Loc, \forall v, v' : Value, \forall x : \texttt{unit} \to GS(A), l \neq l' \to$
   $(\texttt{update}(l, v) \texttt{ >>= } (\lambda\_.(\texttt{update}(l', v') \texttt{ >>= } x))) =$
   $(\texttt{update}(l', v') \texttt{ >>= } (\lambda\_.(\texttt{update}(l, v) \texttt{ >>= } x))).$
10. $\forall A : Set, \forall l, l' : Loc, \forall v, v' : Value, \forall f : Value \to GS(A), l \neq l' \to$
    $(\texttt{update}(l, v) \texttt{ >>= } (\lambda\_.(\texttt{lookup}(l') \texttt{ >>= } (\lambda v'.f\ v')))) =$
    $(\texttt{lookup}(l') \texttt{ >>= } (\lambda v'.(\texttt{update}(l, v) \texttt{ >>= } (\lambda\_.f\ v')))).$

Fig. 1. Equational laws for the global store monad GS.

As usual, for each type A, `GS A` is the type of computations returning values of type A along with the possibly modified store. `ret` embeds a value as a (trivial) computation, and `bind` is used to "concatenate" the effects. We will often abbreviate `(bind A B a f)` as `(a >>= f)`. For any location `l`, `v`, `(update l v)` is the computation which updates the content of `l` with `v`, and `(lookup l)` returns the value in `l`.

This behaviour can be fully specified by means of a set of equational laws, as in [7]. The equational laws are reported in "readable" format in Figure 1; their Coq formalization is in Appendix A.1. These are assumed as `Axiom`s in the module, and should be considered as a specification for any implementation of the state monad. The laws (1)—(3) are the usual ones for monad's unit and multiplication. The remaining seven laws (4)—(10) correspond to Plotkin and Power's axiomatization of a global store semantics [7, Section 3]. It should be noticed that these laws have been adapted to our version of lookup and update operators; e.g., in [7], the lookup operator has type `Loc -> (Value -> GS A) -> GS A` (it takes a location and a continuation, which is "executed" using as input the value contained in the location) whereas our `lookup` has the simpler type `Loc -> GS Value`. We have preferred this formulation because computa-

tio composition is done by the `bind` operator. It is easy to see that the two presentations are equivalent, and hence these laws are Hilbert-Post complete[4].

2.2 Global store monad: internal implementation

The specification provided above is the only informations a user needs in order to reason about programs with global store. In other terms, the equational laws can be taken as `Axiom`s in Coq, with no further details. In order to ensure the consistency of such set of assumptions within the Calculus of Constructions, we provide an internal implementation of this monad: all `Parameter`s are suitably instantiated and all equational laws are proved as `Lemma`ta, by unfolding the definitions of the monadic operators (Figure 2).

Proposition 1. *In the Calculus of Inductive Constructions, the equations in Figure 1 are admissible.*

Clearly this is just one among several possible implementations of the monad specification, and does not necessarily coincide to that of the actual runtime environment in the target language. This is why the user cannot have access to the "internals" of the implementation: she could prove properties which do not hold in the target language by leveraging on peculiar implementation choices (e.g., allocation order of locations, structure of stores, etc.).

3 Extraction example: in-place swap

In this section we give a simple example of specification and extraction of a monadic program with effects. Our aim is to define a program `flip_values` which switches the values of two locations without touching any other location (i.e. the program is *in-place*). A simple type for this program, and its implementation, is the following:

```
Module FlipValues (Mon : GSMonad).
Import Mon.

Lemma flip_values: forall (l l': Loc), GS unit.
Proof.
  intros.
  refine (bind (lookup l) (fun v => bind (lookup l') (fun v' => _ ))).
  eapply bind with (A := unit).
  refine (update l v'). intros. refine (update l' v).
Defined.
```

However, this specification does not tell us anything about the correctness of the program. This can be specified by means an additional result:

[4] An equational theory T is Hilbert-Post complete if every equation in the language is either provable from T or inconsistent with T.

```
Module Import Monad : GSMonad.

Definition Loc := nat.
Definition Value := nat.
Definition Store := Loc -> Value.

Definition ret (A: Set) (x : A): GS A := fun (s: Store) => (x, s).

Definition bind (A B :Set) (a : GS A) (f : A -> @GS B): GS B :=
  fun s : Store => (let (a', s') := (a s) in (f a') s').

Definition update (l: Loc) (n: Value) : GS unit :=
  fun s: Store =>
    let new_state := (fun (l': Loc) => If l = l' then n else s l') in
      (tt, new_state) .

Definition lookup (ref: Loc)  : GS nat :=
  fun (s : Store) => ((s ref), s).

Lemma monad_identity : forall (A: Set) (T: GS A),
  (bind T ret) = T .

Lemma monad_application:
  forall (A B: Set) (f: A -> @GS B) (v: A),
    bind (ret v) f = f v.

Lemma lookup_update_idempotence:
  forall (A : Set) (l : Loc) (x : GS A),
    bind (lookup l) (fun v => bind (update l v) (fun _ => x)) = x.

...
```

Fig. 2. Internal implementation of the store monad.

```
Lemma flip_values_is_in_place:
  forall (l l': Loc),
    { b :  GS unit |
      (bind b (fun _ => bind (lookup l') (fun v => ret v))) =
         (bind (lookup l) (fun v => bind b (fun _ => ret v)))
      /\
      (bind b (fun _ => (bind (lookup l) (fun v => ret v)))) =
        (bind (lookup l') (fun v => bind b (fun  _ => ret v)))
      /\
      forall l'', l'' <> l -> l'' <> l' ->
        (bind b (fun _ => bind (lookup l'') (fun v => ret v))) =
          (bind (lookup l'') (fun v => bind b (fun _ => ret v)))}.
Proof.
```

```
intros.
exists (flip_values l l').
unfold flip_values.  splits_all.
...
Qed.
```

The specification consists of three equational clauses. The first two clauses state that the values in `l` and `l'` are swapped; the third states that all other locations are left unchanged. These clauses (and hence the whole lemma) can be proved just by using the equational laws in Figure 1, without knowing the actual implementation of the operators.

Using the `Extraction` facility offered by Coq, we obtain the following ML code:

```
(** val flip_values :
    Loc -> Loc -> unit GS **)

let flip_values l l' =
  bind (lookup l) (fun v ->
    bind (lookup l') (fun v' ->
      bind (update l v') (fun h -> update l' v)))

let flip_values_in_place l l' = flip_values l l'
```

Notice that this program is within the purely functional fragment of ML, extended with constants `bind` , `ret`, etc, representing the monadic operators. These are not implemented, so the program cannot be executed in a ML runtime environment as it is. The next step will be to implement the back-end compiler which replaces these abstract operators with the corresponding code, so that the program can be effectively run.

4 From intermediate to target language

In this section we describe the back-end compiler from the intermediate language, i.e., the language covered by the extraction mechanism, to the target language. The intermediate language is (a functional fragment of) either Objective CaML, Haskell or Scheme, extended with monad's four constructors. For sake of simplicity, we have chosen ML as the basis for both the intermediate and the target language, so that the compiler is simpler. However, the approach holds for any target language with some representation of store, as long as it can be given a formal semantics in Coq.

Target language The abstract syntax of the target language is the following:

$$E ::= \ x \mid n \mid \text{let } x := E \text{ in } E \mid (E, E) \mid \pi_1 E \mid \pi_2 \ E \mid EE \mid \lambda x.E \mid E := E \mid !E$$

For the formalization of this language, we adopt Charguéraud's *locally nameless* implementation [2]. We refer to the website[5] for the details; we mention only that the modules `ML_Definitions` and `ML_Infrastructure` provide the data structures and main results for a fragment of ML with data types, recursion, references and exceptions. The basic definitions we need are the following:

- `trm : Set` is the grammar of terms; some constructors are the following

```
Inductive trm : Set :=
  | trm_bvar  : nat -> nat -> trm
  | trm_fvar  : var -> trm
  | trm_abs   : trm -> trm
  | trm_fix   : trm -> trm
  | trm_let   : trm -> trm -> trm
  | trm_app   : trm -> trm -> trm
...
  | trm_loc   : loc -> trm
  | trm_ref   : trm -> trm
  | trm_get   : trm -> trm
  | trm_set   : trm -> trm -> trm
...
```

- `Inductive term : trm -> Prop` is the well-formedness judgments of locally-closed pre-terms (i.e., possibly untyped terms, but without free de Bruijn indexes);
- `Definition sto := LibEnv.env trm.` is the store (i.e., a list of terms);
- `Inductive sto_ok : sto -> Prop` is the well-formedness judgments of stores;
- `Inductive value : trm -> Prop` defines the subset of values;
- `Definition conf := (trm * sto)` is the set of machine configurations (i.e., term-store pairs);
- `Inductive reds : conf -> conf -> Prop` is the big-step semantics.

Intermediate language On the other hand, the intermediate language, which we call *Computational ML*, is formalized in the module `CompML`. Here we use again the locally nameless syntax for dealing with binders.

```
Module CompML.
...
Inductive trm : Type :=
| trm_bvar  : nat -> nat -> trm
| trm_fvar  : var -> trm
| trm_abs   : trm -> trm
| trm_fix   : trm -> trm
| trm_let   : trm -> trm -> trm
| trm_app   : trm -> trm -> trm
| trm_unit  : trm
```

[5] `http://www.chargueraud.org/softs/ln/`

```
| trm_nat   : nat -> trm
| trm_pair  : trm -> trm -> trm
| trm_inj1  : trm -> trm
| trm_inj2  : trm -> trm
| trm_loc   : loc -> trm
| trm_ret   : trm -> trm
| trm_bind  : trm -> trm -> trm
| trm_lookup: trm -> trm
| trm_update: trm -> trm -> trm.
```

As mentioned before, this language is a functional subset of ML, extended with the monad constructors (the last four constructors). Over this data type we define some standard judgments and functions for dealing with the locally nameless syntax, similarly to the target language; in particular, we have the well-formedness judgments of locally-closed pre-term `term : trm -> Prop`. We do not give this language an operational semantics: the behaviour we are interested in is completely specified by the equational laws in Figure 1.

Back-end compiler The back-end compiler is formalized as a function from `CompML.trm` to `ML.trm`. Basically, it replaces `trm_ret` with nothing, `trm_bind` with `trm_let`, `trm_lookup` with `trm_get` and `trm_update` with `trm_set`.

```
Fixpoint compile (e: CompML.trm) : trm :=
  match e with
    | CompML.trm_bvar n n' => trm_bvar n n'
    | CompML.trm_fvar n => trm_fvar n
    | CompML.trm_loc l  => trm_loc l
    | CompML.trm_let t t' => trm_let (compile t) (compile t')
    | CompML.trm_app e1 e2 => trm_app (compile e1) (compile e2)
    | CompML.trm_ret t  => compile t
    | CompML.trm_bind t t' => trm_let (compile t) (compile t')
    | CompML.trm_lookup t => trm_get (compile t)
    | CompML.trm_update t t'=>  trm_set (compile t) (compile t')
...
  end.
```

This translation is almost trivial because CompML and the target language we have chosen are very close. However, we can apply this approach to any other language with some representation of stores (even not functional). In general, we can we may need to replace each monadic constructor with longer code snippets, as we did in [4]. The more different from ML the target language, the more complex the compiler.

Correctness of the compiler In order to prove the correctness of the compiler, we have to show that all equational laws in Figure 1 are respected. More precisely, for each equational law of the form $e_1 = e_2$, the terms $e_1, e_2 : GS\ A$ are built using the monadic constructors; what we have to prove is that the corresponding

programs extracted by Coq are *semantically equivalent*. The notion of semantic equivalence is lifted from the operational semantics of the target language, and hence depends on the specific computational aspects we are considering. In our case, we have to introduce a semantic equivalence for ML programs, taking into account that it is an imperative language.

Definition 1. *We say that two CompML programs* e_1, e_2 *are* semantically equivalent *(*$e_1 \approx e_2$*) iff the corresponding compiled programs, starting from any well-formed store* m*, either both diverge, or both converge to two equivalent configurations* (t, m_1), (t, m_2)*, i.e.* m_1 *and* m_2 *are extensionally equal (they agree on all locations).*

Formally, this is rendered as follows (we omit the definition of `used_loc`):

```
Definition subset_mem (l: list loc) (m: sto): Prop:=
    forall (x: loc), In x l ->
      exists (v: trm), binds x v m /\ exists n, trm_nat n = v.

Definition reduce c1 c2 :=
  forall t s, reds c1 (t, s) ->
    exists s', reds c2 (t,s') /\ memory_eq s s'.

Definition prog_equiv c1 c2 := reduce c1 c2 /\ reduce c2 c1.

Definition sem_eq (e1: CompML.trm) (e2: CompML.trm) : Set :=
  forall (m: sto),
    (subset_mem (used_loc e1) m) /\ (subset_mem (used_loc e2) m) /\
    sto_ok m -> prog_equiv (compile e1, m) (compile e2, m).
```

Then, for each equational law in Figure 1, we can state (and prove) a corresponding equivalence result. Here we list some examples.

```
Lemma monad_identity : forall t,
  sem_eq (CompML.trm_bind t (CompML.trm_ret (CompML.trm_bvar 0 0))) t.

Lemma monad_composition : forall x f g,
  CompML.term (CompML.trm_bind (CompML.trm_bind x f) g) ->
  sem_eq (CompML.trm_bind (CompML.trm_bind x f) g)
         (CompML.trm_bind x (CompML.trm_bind f g)).

Lemma lookup_after_update: forall (l:loc) (f: CompML.trm) v,
    CompML.term f ->
  sem_eq ((CompML.trm_bind
           (CompML.trm_update (CompML.trm_loc l) (CompML.trm_nat n))
           (CompML.trm_bind
             (CompML.trm_lookup (CompML.trm_loc l))
               (CompML.trm_app f (CompML.trm_bvar 0 0)))))
```

```
          (CompML.trm_bind
            (CompML.trm_update (CompML.trm_loc l) (CompML.trm_nat n))
            (CompML.trm_app f (CompML.trm_nat n))).
...
```

All these lemmata can be proved by unfolding `sem_eq`, applying the `compile` function and then reasoning over the semantics of the resulting ML program. Therefore, we have proved the following:

Proposition 2. *The function* `compile` *respects the equational laws of the store monad; hence, for all terms* `t1, t2` *such that* `t1 = t2` *is derivable in Coq extended with the axioms in Figure 1, it is* `(sem_eq (compile t1) (compile t2))`.

This result allows us to define the back-end compiler from the concrete syntax of extracted code to the concrete syntax of ML, just by mimiking the action of `compile`. Applying this compiler to the code of `flip_values` of Section 3, we obtain the following code, which can be executed in a ML runtime environment:

```
let flip_values l l' =
  let v = !l in
    let v' = !l' in
      let h = (l := v') in
        l' := v.
```

In virtue of Proposition 2, this program is certified to satisfy its specification, given in `flip_values_is_in_place`.

5 Conclusions

In this paper we have presented a methodology for the synthesis of *certified programs with effects* taking advantage of Coq `Extraction` facility, extending the work initiated in [4]. This methodology can be summarized as follows:

1. Give a monad specification as a `Module Type` MT in Coq, where all the data structures and operators are declared as `Parameter`s, and their behaviour is specified by a set of `Axiom`s.
2. Prove that these axioms are sound, by providing an internal implementation, i.e., a `Module` M_{int} implementing the `Module Type` MT above.
3. The `Extraction` mechanism produces programs in a *computational meta-language* CompML, which is a functional language extended with constants representing the monad's operator.
4. Implement a back-end compiler from CompML to a target "real-world" language. Basically this means to define how each parameter and each constructor specified in the `MT` monad type is actually implemented in the target language.
5. Formalize the syntax of the language CompML in Coq, e.g. using *locally nameless* abstract syntax (this is easy, since the language is always the same functional core, plus the monad's constructors).

6. Formalize the syntax and the semantics of the target language in Coq, and formalize the back-end compiler as a recursive function from CompML to this language.
7. Prove the soundness of the back-end compiler by proving that it respects all *Axiom*s in the monad specification MT.

At this point, program specifications and constructive proofs can be carried out within a module parameterized on MT, i.e. `Module Programs (M:MT). Import M.` A typical format is the following: `forall x:A, {y:(T B) | P(x,y)}.` where `T` is the monad. These proofs can use the *Axiom*s about the monad operators declared in the module type MT (but not the details of the internal implementation M_{int}). The code obtained by `Extraction` is in CompML, which can be fed to the back-end compiler to obtain certified code which can be effectively executed in the target environment.

In this paper, we have applied this methodology to the *global store* monad:

- its specification `GSMonad` is based on Plotkin-Power's sound and complete axiomatization [7];
- its internal implementation is based on a simple representation of stores as functions from locations to values;
- as target language, we have chose ML with references, so that the back-end compiler is very simple;
- for the formalization of the target language, we have adopted Charguéraud's implementation of Mini-ML [2].

As an example, we have provided the specification of the in-place "location swap" function, proved its correctness, and extracted its certified code. (As a side result, we have given a formal proof that Charguéraud's implementation is sound with respect to the global store semantics.)

There are several directions for further work. First, we would like to apply this methodology to other computational aspects. In general the main obstacle is to have sound and complete axiomatizations of the corresponding monads, and a formalization in Coq of the target language. A good candidate is the monad for the *local store*, whose axiomatization is provided in [7] as well, and whose target language is still ML with references. This example would be interesting also beacause it would lead us to consider a monad over a *presheaf category*, instead of Set. Other interesting aspects are distributed computations (with some distributed languages as target, e.g. Erlang), probabilistic computations, etc.

Another interesting issue is about the "usability" of the equational laws, like those for the global store monad. In fact, proving properties about programs to be extracted using only these laws can be long and tedious (e.g., proving the specification of the "flip value" example still requires some hundreds steps). It should be possible to lighten this burden by developing domain-specific, *ad hoc* logics on top of the equational laws of the monad specification, along with a set of tactics. For instance, from the equational theory for the global store monad we can derive a Hoare logic, possibly based on separation logic [8]. Hoare triples are somewhat simpler to use than equational laws, and new rules can be

added as needed provided that they do not break the equational laws. Other computational monads would lead to different logics (e.g., temporal, dynamic, probabilistic...).

References

1. J. Armstrong. Erlang - a survey of the language and its industrial applications. In *Proceedings of the symposium on industrial applications of Prolog (INAP'96)*, 1996.
2. A. Charguéraud. The locally nameless representation. *J. Autom. Reasoning*, 49(3):363–408, 2012.
3. P. Letouzey. Extraction in Coq: An overview. In A. Beckmann, C. Dimitracopoulos, and B. Löwe, editors, *Proc. CiE*, volume 5028 of *Lecture Notes in Computer Science*, pages 359–369. Springer, 2008.
4. M. Miculan and M. Paviotti. Synthesis of distributed mobile programs using monadic types in coq. In L. Beringer and A. Felty, editors, *Proc. ITP'12*, Lecture Notes in Computer Science. Springer, 2012.
5. E. Moggi. Notions of computation and monads. *Information and Computation*, 1, 1993.
6. C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.
7. G. D. Plotkin and J. Power. Notions of computation determine monads. In M. Nielsen and U. Engberg, editors, *Proc. FoSSaCS*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.
8. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS*, pages 55–74. IEEE Computer Society, 2002.

A Some Coq code

A.1 Equational laws for the Global Store monad

```
(* Laws for the monad unit and multiplication *)
Axiom monad_identity : forall (A: Set) (t: @GS A),
   (bind t (fun v => (ret v))) = t.
Axiom monad_application:
  forall (A B: Set) (f: A -> @GS B) (v: A),
    bind (ret v) f = let x:=v in (f x).
Axiom monad_composition:
  forall (A B C: Set) (f: A -> @GS B) (g: B -> @GS C) (x: @GS A),
    bind (bind x f) g = bind x (fun y => bind (f y) g).

(* Laws for the lookup and update operations *)
Axiom lookup_update_idempotence:
  forall (A : Set) (l : Loc) (x : @GS A),
      bind (lookup l) (fun v => bind (update l v) (fun _ => x)) =  x.
Axiom lookup_idempotence:
  forall (A B:Set) (l : Loc ) (f : Value -> Value -> @GS A),
  (bind (lookup l) (fun x => bind (lookup l) (fun y => (f x y))))  =
```

```
  (bind (lookup l) (fun x => (f x x))).
Axiom update_idempotence:
  forall (A: Set) (l: Loc) (v v': Value) (x: unit -> GS A),
  (bind (update l v) (fun _ => bind (update l v') x)) = bind (update l v') x.
Axiom lookup_after_update:
  forall (A B: Set) (l: Loc ) (v v': Value) (f: Value -> @GS A),
  bind (update l v) (fun _ => (bind (lookup l) (fun v' => f v'))) =
  bind (update l v) (fun _ =>  (f v)).
Axiom lookup_commutativity:
  forall (A: Set) (l l': Loc ) (f: Value -> Value -> @GS A),
    l <> l' ->
    bind (lookup l) (fun v => (bind (lookup l') (fun v' => (f v v')))) =
    bind (lookup l') (fun v' => (bind (lookup l) (fun v => (f v v')))).
Axiom update_commutativity:
  forall (A: Set) (l l': Loc) (v v': Value) (x: unit -> GS A),
    l <> l' ->
    bind (update l v) (fun _ => (bind (update l' v') x)) =
    bind (update l' v') (fun _ => bind (update l v) x).
Axiom update_lookup_commutativity:
  forall (A: Set) (l l': Loc ) (v v': Value) (f: Value -> @GS A),
    l <> l' ->
    bind (update l v) (fun _ => bind (lookup l') (fun v' => f v')) =
    bind (lookup l') (fun v' => bind (update l  v) (fun _ => f v')).
```