Synthesis of distributed mobile programs using monadic types in Coq*

Marino Miculan Marco Paviotti

Dept. of Mathematics and Computer Science, University of Udine, Italy.
marino.miculan@uniud.it, marco.paviotti@gmail.com

Abstract. We present a methodology for the automatic synthesis of certified, distributed, mobile programs with side effects in Erlang, using the Coq proof assistant.
First, we define monadic types in the Calculus of Inductive Constructions, using a lax monad covering the distributed computational aspects. These types can be used for the specifications of programs in Coq. From the (constructive) proofs of these specifications we can extract Haskell code, which is decorated with symbols representing distributed nodes and specific operations for distributed computations. These syntactic annotations are exploited by a *back-end* compiler to produce actual mobile code for a suitable runtime environment (Erlang, in our case).
Then, we introduce an object type theory for distributed computations, which can be used as a front-end programming language. These types and terms are translate to CIC extended with monadic types; this allows us to prove the soundess of the object type theory, and to obtain an implementation of the language via Coq's extraction features.
This methodology can be ported to other computational aspects, by suitably adapting the monadic type theory and the back-end compiler.

1 Introduction

One of the most interesting features of type-theory based proof assistants, like Coq, is the possibility of extracting programs from proofs [9,6]. The main motivation for this extraction mechanism is to produce certified programs: each property proved in Coq (e.g., a precise specification between input and output of programs) will still be valid after extraction. Since the extraction mechanism relies on the well-known Curry-Howard "proofs-as-programs", "propositions-as-types" isomorphism, the extracted programs are naturally expressed in functional languages such as OCaml, Haskell or Scheme.

However, there are many other computational aspects that a programmer has to deal with, and whose programs are quite difficult to reason on; e.g., distributed concurrent programs, web services, mobile code, etc. These scenarios would greatly benefit from mechanisms for extraction and automatic synthesis of certified programs. Unfortunately, the programming languages implementing these aspects usually do not feature a rich type theory supporting a Curry-Howard isomorphism. Even if such a theory were available, implementing a specific proof-assistant with its own extraction facilities would be a daunting task.

* Work supported by MIUR PRIN project 20088HXMYN, "SisteR".

In this paper, we propose a methodology for circumventing this problem using the extraction mechanisms of existing proof assistants, namely Coq. Basically, the idea is that the proof assistant's type theory (e.g., the Calculus of Inductive Constructions) can be extended with a suitable computational monad `IO : 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 `Proposition`s to specify the types of (non-functional) programs, like e.g., `f:A -> IO B`. These propositions can be proved constructively as usual, possibly using the specific properties of the monad operators. From these proofs we can extract functional programs (e.g., in Haskell) by taking advantage of the standard Coq extraction facility. In general these programs cannot be immediately executed because the functional language may not support the specific computational feature we are dealing with. Nevertheless, we can cover this gap by implementing a suitable "post-extraction" translation from Haskell to a suitable target language, with the required features. In fact, the non-functional features in the extracted Haskell programs are represented by the constructors of the computational monad, and these informations can be easily exploited by the post-extraction translation to generate the target program. Thus, Haskell can be seen as an intermediate language, and the post-extraction translation is technically a *back-end compiler*.

Overall, this methodology can be summarized in the following steps:

1. Define a `IO` monad over `Set` in Coq, with the required constructors covering the intended computational aspects.
2. Implement the back-end compiler from Haskell to the target language. Basically this means to define how each constructor of the `IO` monad is actually implemented in the target language. These implementations have to respect the assumptions/the reference implementations.
3. State and constructively prove propositions over types of the form `IO A`. A typical format is the following: `forall x:A, {y:(IO B) | P(x,y)}`. These proofs can use the properties of the monad operators.
4. Extract the Haskell function from the proof, and use the back-end compiler to translate this code into the target language.
5. Execute in the target environment the program obtained in this way.

This approach is quite general and powerful, as it can be effectively used for programming in Coq in different notions of computational. However, it requires the user to have a non trivial knowledge of the Coq proof system. In fact, a programmer may prefer to use a high-level language, possibly specialized *ad hoc* for some particular aspects. Being more specialized, these *front-end* languages are not as expressive as the Calculus of Inductive Constructions, but simpler to use. Still we can consider to translate these languages into Coq, by means of a *front-end compiler*: each syntactic type of the object language is interpreted as a `Set` (possibly using the monad), and terms of the object syntax denote CIC terms between these `Set`s. Thus, CIC extended with the computational monad is used as a framework for the *semantic* interpretation of a given object type theory. This allows to take advantage of Coq type checking for checking object

level terms; moreover, we can obtain readily and implementation of the front-end language by means of the Coq extraction mechanisms and the backend compiler.

Hence, we can extend the above methodology with the following steps:

6. Define a front-end language, i.e., an *object* theory of types and terms. Non-functional computational aspects should be covered by a lax modality.
7. Formalize in Coq the object type theory, the terms, and the typing judgment, via a deep encoding (i.e., as `Inductive Set`s).
8. Define a translation of the object types to Coq `Set`s. In particular, object types with lax modality are interpreted to sets of the form `IO A`.
9. Prove that the type theory is consistent w.r.t. the semantics, giving a translation of well-typed object terms to functions among the corresponding `Set`s.

At this point, we can specify a program using the object type theory; translate this specification into a Coq `Proposition`; prove it as usual; extract the Haskell program, and compile it into the target language. Or we can write a program directly in the object term language, translate it to a Coq proof via the soundess result, extract the Haskell program and compile it to the target language.

In the rest of this paper we will apply this methodology for the synthesis of distributed, mobile programs in Erlang [1,11]. This is particularly relevant since Erlang is an *untyped* language, with little or no support for avoiding many programming errors. On the other hand Erlang offers excellent support for distributed, mobile computations.

First, in Section 2 we introduce the "semantic" monad `IO` in Coq, with corresponding constructors covering the intended computational aspects (namely, code mobility, side-effects and errors). In Section 3 we describe the back-end compiler from Haskell to Erlang, together with a complete working (although quite simple) example. Then, in Section 4 we will define λ_{XD}, a type theory for distributed computations with effects similar to Licata and Harper's HL5 [7], together with its formalization in Coq (using both weak HOAS and de Bruijn indexes at once). We show how these syntactic datatypes and terms are translated into Coq `Set`s (using the `IO` monad) and functions, thus proving the soundness of the object type theory. As a not-so-trivial example, in Section 5 we give a complete specification (in λ_{XD}) and extraction of remote read/write operations. Concluding remarks and directions for further work are in Section 6.

The Coq and Haskell code of the whole development, with examples, is available at `http://sole.dimi.uniud.it/~marino.miculan/LambdaXD/` .

2 Monadic types for distributed computations in Coq

In order to model distributed computation with effects we need to define what a store is, and take into account that we are in a distributed scenario. To this end we define a monad (actually, a world-indexed family of monads) covering all non-functional computational aspects: distributed dependency of resources and memory operations and side effects, possibly with failure.

The type `Store` models distributed stores. Among several possibilites, the simplest is to keep a *global store* which contains all the memory locations of

every host. Let us assume to have a type `world` of "worlds", i.e., host identifiers, and for simplicity that locations and values can be only natural numbers. Then, a store is a list of triples "(world, location, value)":

```
Inductive Ref (w : world): Set :=  Loc : nat -> Ref w.
Inductive Store : Set :=
 Empty : Store | SCons : (world * (nat * nat)) -> Store -> Store.
```

We can now define the family of monads: given a world w, for each type A the type `IO w A` is the type of computations returning values in A on world w:

```
Definition Result (w: world) (A: Set): Set := A * Store.
Definition IO (w : world) (A : Set) : Set :=
   Store -> option (Result w A).
```

where `option` is the counterpart of Haskell's `Maybe`. Thus a computation yields either an error or a value of type `Result w A` which contains a value of type A, localized at w, and the new store. Thus, a computation of type `(IO w A)` carries the stores of all worlds, not only w. This is needed for allowing a computation at w to execute computations on other worlds, possibly modifying their stores. As a consequence, a term of type $(\mathsf{Ref} \to \bigcirc A)$ `<w>` actually is a function of type `(Ref w) -> (IO w` A`<w>)`: the argument must be a reference local to world w.

Now we have to define the constructors for the monadic types. The first two constructors are those of any monad, namely "return" and "bind". `IOret` embeds a value as a computation, i.e., a function from states to results; `IObind` "concatenates" the effects.

```
Definition IOret (w : world) (A: Set) (x : A): IO w A :=
  fun (s: Store) => Some (pair x s).
Definition IObind (w:world)(A B:Type)(a:IO w A)(f:A -> IO w B):IO w B :=
  fun s : Store =>
    match (a s) with Some (pair a' s') => (f a') s' | None => None  end.
```

Then, the state is manipulated by the usual "lookup" and "update":

```
Definition IOlookup (w:world)(A:Set):Ref w -> (nat -> IO w A) -> IO w A:=
  fun addr f s =>
    match (do_seek_ref s addr) with Some result => f result s
                                  | None => None   end.
Definition IOupdate (w: world) (A: Set):
Ref w -> nat -> IO w A -> IO w A :=
  fun addr v m s =>  match m s with
                     Some (result, result_state) =>
                        let r := (do_update w result_state addr v) in
                          match r with None => None
                                 |  Some s' => Some (pair result s')
                          end
                   | None => None
    end.
```

where `do_seek_ref` and `do_update` are suitable auxiliary functions. Basically, `(IOlookup w A n f)` first finds the value `v` associated to location n on world w,

then executes the computation `f v`; similarly, `(IOupdate w A n v M)` executes M in the state obtaned by updating the location n with the value v. (For sake of simplicity, all references are given a different location number, even if they reside on different worlds.)

New locations are generated by means of `IOnew`, which adds a new local reference at the top of the state, before executing the continuation:

```
Definition IOnew (w:world)(A:Set): nat -> (Ref w -> IO w A) -> IO w A :=
 fun (x : nat) (f: Ref w -> IO w A) (s : State) =>
   let location := (Loc w (IOfree_index w s)) in
     let new_state := (SCons (pair (pair w (IOfree_index w s)) x) s) in
       f location new_state.
```

For modeling distributed computations, we add a function allowing for remote executions and retrieving the value as a local result.

```
Definition IOget (w remote: world) (A: Type) : (IO remote A) -> IO w A :=
  fun (a : IO remote A) (s : Store) => (a s)
```

Given a state `s`, `IOget` function executes the remote computation by giving it the state and returning the same object but in a different world. This could look strange, as the result is untouched—but the type is different, since it becomes `IO w A` (automatically inferred by Coq typing system). Notice that we can pass the remote computation the state of a local one, because each `s: Store` contains the locations and corresponding values for every world.

Remark 1. We have given an explicit definition of the data structures and operations needed for the `IO` monad. A more abstract approach would be to define the operations of the monads as atomic constructors of an abstract data type, together with suitable equational laws defining their behaviour:

```
Record Monad := mkMonad {
  IO : world -> Type -> Type;
  IOreturn : forall w A, A -> (IO w A);
  IObind   : forall w A B, (IO w B) -> (B -> (IO w A)) -> (IO w A);
  IOlookup : forall w A, Ref w -> (nat -> IO w A) -> (IO w A);
  IOupdate : forall w A, Ref w -> nat -> (IO w A) -> (IO w A);
  IO_H1 : forall w A a f, (IObind w A A (IOreturn w A a) f) = (f a);
  ...
}.
```

This approach would be cleaner, since it abstracts from the particular implementation. However, in order to be able to prove all properties we may be interested in, we need to know an equational theory (more precisely, a (Σ, E) algebraic specification) complete with respect to the intended computational aspects. This is a not trivial requirement. Power and Plotkin have provided in [10] a complete specification for the state monad, but we do not know of a similar complete axiomatizations for distributed computations, nor if and how this would "merge" with the former. Hence, we preferred to give an explicit definition for each operation: on one hand, the required properties can be proved instead to be assumed, and on the other these definitions have to be intended as "reference implementations" of the computational aspects.

3 Extraction of distributed programs

The monadic types described above, can be used for specifying programs for mobile distributed computation, as in the following example:

Example 1 (Remote procedure call). We prove a Lemma stating that a procedure on a world `server` can be seen as a procedure local to a world `client`:

```
Lemma rpc : forall client server, forall A,
            (nat -> (IO server A))  ->  (nat -> (IO client A)).
intros f n; eapply IOget; apply f; assumption.Qed.
```

From the proof of this Lemma, using the *Recursive Extraction* command we obtain a Haskell program (together with all functions and datatypes involved):

```
rpc :: world -> world -> (Nat -> IO a1) -> Nat -> IO a1
rpc client server f n =  iOget client server (f n)
```

The type of the function extracted from the proof of `rpc` is essentially the same as of the specification, and its body is stripped of all logical parts. Notice that monadic constructors (e.g., `IOget`) are not unfolded in the extracted code, which therefore contains undefined symbols (`iOget`). We could define these symbols directly in Haskell, but this would be cumbersome and awkward due the distributed aspects we have to deal with. In this section, we discuss how this Haskell code is turned into a distributed Erlang program, by means of a "back-end compiler". (For an introduction to Erlang, we refer to [1,11].)

Let us denote by $(\!|\cdot|\!)_\rho$ the translating function from Haskell to Erlang, where ρ is the environment containing the Erlang module name and, for each identifier, a type telling whether this is a function or a variable, and in the former case the arity of that function. This is needed because in Haskell function variables begin with a capital letter, while in Erlang only variables do; the arity is needed to circumvent the currying of Haskell, which is not available in Erlang.

Most of Haskell syntax is translated into Erlang as expected, both being functional languages (but with different evaluation strategies). Monad operations need special care: each IO function must be implemented by a suitable code snippet, conforming to the intended meaning as given by their definition in Coq. One may think of Coq definitions as "pseudo-" or "high-level" code, and the Erlang implementations as the actual, "low level" code.

Implementation of mobility: IOget An application of the *IOget* constructor is extracted as a Haskell term (`iOget` A_1 A_2 $(F\ A_3)$), where A_1 is the actual world, A_2 is the remote world, and F is the term to be remotely evaluated with parameters A_3. This term is translated in Erlang as follows:

$$
\begin{aligned}
(\!|\texttt{iOget}\ A_1\ A_2\ (F\ A_3)|\!)_\rho = \textbf{spawn}&\texttt{(element(2, } (\!|A_1|\!)_\rho\texttt{), } \rho\texttt{("modulename"),}\\
&\texttt{dispatcher,[fun () -> } (\!|F|\!)_\rho (\!|A_3|\!)_\rho \textbf{end},\\
&(\!|A_2|\!)_\rho, \{\textbf{self}\texttt{(), }\textbf{node}\texttt{()}\}\texttt{])},\\
\textbf{receive}&\{\texttt{result, Z -> Z}\}
\end{aligned}
$$

Basically, this code **spawn**s a process on the remote host, sending it the code to be executed, and waits for the result with a synchronous **receive**. Let us examine the arguments of **spawn**. The first argument is the host address we are going to send the code to; in fact, in Erlang a process is identified by the pair `(pid, host-address)`, and hence we use these pairs as the implementations of λ_{XD} worlds. The second and third arguments are the module name and the function name to execute, respectively, which we will describe below. The fourth argument is a triple whose first component is the code to be executed applied to its parameters. Since Erlang is call-by-value, we have to suspend the evaluation of the program before sending to the remote site; to this end the application is packed in a vacuous λ-abstraction. The second and third components are the address of the final target (i.e., where the computation must be executed) and of the local node (i.e., the process which the result must be sent back to).

The function we spawn remotely is `dispatcher`, which takes a function and send it to another host where there is an `executer` function listening:

```
dispatcher(Mod, Fun, Executer, Target) ->
    spawn(element(2,Executer),update,executer,[Mod,Fun,Target]).
executer(Mod, Fun, FinalHost) ->
    Z = Fun(),  element(1,FinalHost) ! {result, Z}.
```

The dispatcher behaves as a code forwarder: it spawns a new executer process on the Executer machine passing it the code and the final target where the result has to be sent back. When the executer function receives the module, the function to execute and the target host, it simply evaluates the function by applying to (); then the result is sent back to the caller.

Implementation of references: IOnew, IOupdate, IOlookup Being a declarative language, Erlang does not feature "imperative-style" variables and references. Nevertheless, we can represent a location as an infinite looping process which retains a value and replies to "read" messages by providing the value, and to "update" messages by re-starting with the new value:

```
location(X) -> receive
                {update, Val} -> location(Val);
                {get, Node} -> element(1, Node) ! {result, X}, location(X)
               end.
```

Therefore, creating a new location at a given world is simply spawning a new `location` process on the host corresponding to that world. The spawning primitive embedded in Erlang returns the pid of the spawned process which is passed to the continuation program. This is implemented in the translation of the `iOnew` function (which is extracted from the `IOnew` monad constructor):

$$(\!|\texttt{iOnew}\ A_1\ A_2\ A_3|\!)_\rho = ((|\!|A_3|\!|)_\rho)(\textbf{spawn}\,\textbf{(element}\,\texttt{(2,}\ (|\!|A_1|\!|)_\rho), \rho\texttt{("module name"),}\ \texttt{location, [}(|\!|A_2|\!|)_\rho\texttt{])})$$

On the other hand, the implementation of `IOupdate` sends an "update" message to process whose pid is A_1, so it updates its internal value and continues

with continuation program which takes no argument:

$$(\!|\texttt{iOupdate}\ w\ A_1\ A_2\ A_3|\!)_\rho = (\!|A_1|\!)_\rho!\{\mathbf{update}, (\!|A_2|\!)_\rho\}, (\!|A_3|\!)_\rho$$

Notice that the Erlang typing discipline does not ensure either that A_1 is a pid referring to a local process (i.e., a local location), or that it is actually executing `location/1`. However, both these properties are guaranteed by Coq typing rules.

Finally the implementation of `IOlookup` is quite similar to `IOupdate`'s: the translation sends a "get" message, along with the caller address in order to get the response, and synchronously waits for the answer. The result of the operation is passed to the rest of the program:

$$\begin{aligned}(\!|\texttt{iOlookup}\ w\ A_1\ A_2|\!)_\rho = (\!|A_1|\!)_\rho!\{\texttt{get}, \{\mathbf{self}(), \mathbf{node}()\}\},\\ \mathbf{receive}\{\texttt{result, Z}\} \to ((\!|A_2|\!)_\rho)(Z)\end{aligned}$$

Example 2. Let us consider the remote procedure call function of Example 1. From that Haskell code, we can extract an Erlang procedure for the remote execution of any program that takes a natural, running our Haskell-to-Erlang compiler. We obtain the following Erlang program:

```
rpc (Client, Server, F, N)->
    spawn (element (2, Client), rpc, dispatcher, [rpc,
  (fun () -> F (N)end),
  Server,
  { self (), node () } ]),
    receive { result, Z } -> Z end .
```

As expected, the extracted program has been translated into a spawn function which passes the term `X(H)` to the dispatcher on `w`, which in turn executes the function at `w'` by spawning a remote process; then it receives the result from the server, and finally returns it.

4 A type theory for distributed computations

The monadic type theory we have presented above can be used for specifying and synthesizing Erlang distributed programs, but it requires the user to have a non trivial knowledge of the Coq proof system. In fact, a programmer may prefer to use a high-level language, possibly focused on some particular aspects (e.g., web service orchestration, code mobility, etc.). Being more specialized, these *front end* languages may be not as expressive as the Coq internal logic, but simpler to use. Nevertheless, we can consider to translate these languages into CIC for taking advantage of the type checking and program extraction features of Coq.

As an example of this approach, in this section we present a simple type theory for distributed computation with references, called λ_{XD} ("lambda-cross-D"), and give it an interpretation in Coq using the monadic type theory above.

4.1 λ_{XD}

The theory λ_{XD} is similar to Licata and Harper's HL5 [7], the main difference is that we have an explicit language for terms. This type theory has hybrid modal constructors for dealing with *worlds*, mobile computations, and references.

Syntax The syntax of types and terms is defined as follows.

$$
\begin{array}{lll}
\textit{Types} & A ::= & \mathsf{Unit} \mid \mathsf{Bool} \mid \mathsf{Nat} \mid \mathsf{Ref} \\
& & \mid A \times B \mid A \to B \mid \forall w.A \mid \exists w.A \mid A@w \mid \bigcirc A \\
\textit{Terms} & N, M ::= & x \mid n \mid \mathsf{true} \mid \mathsf{false} \\
& & \mid \mathsf{return}\ M \mid \mathsf{get}\ \mathsf{w}\ M \mid \mathsf{bind}\ MN \mid \mathsf{assign}\ MN \mid \mathsf{new}\ M \\
& & \mid \mathsf{deref}\ M \mid (M, N) \mid \pi_1\ M \mid \pi_2\ M \mid MN \mid \lambda x.M \\
& & \mid \mathsf{hold}\ M \mid \mathsf{leta}\ x = M\ \mathsf{in}\ N \mid \mathsf{letd}\ (w, x) = M\ \mathsf{in}\ N \\
& & \mid \mathsf{some}\ w\ M \mid \Lambda w.M \mid \mathsf{unbox}\ w\ M
\end{array}
$$

The types $\to$ and $\times$ represent functions and products. The types $\forall$ and $\exists$ represent quantifiers over worlds. Next, @ is a connective of hybrid logic, which allows to be set the world at which a type must be interpreted. Finally, $\bigcirc$ and Ref represent monadic computations and references; note that they are not indexed by a world. The usual modal connectors $\Box$ and $\Diamond$ can be defined using quantifiers and @: $\Box A = \forall w.A@w$ and $\Diamond A = \exists w.A@w$.

Regarding terms, we have variables, numbers and booleans, etc. Constructors λ, leta, letd, Λ bind their variables as usual. leta is the usual "let" constructor, which binds x to the value of M, then evaluates N; letd is the elimination of existential types: M is expected to evaluate to some $u\ V$, and w, x are bound to u, V before evaluating N. Notice that worlds are not first class objects (this would lead to computations accepting and returning worlds, with a correspondingly more complex type theory). For monadic types we have the standard monadic constructors return and bind, plus constructors for *local* state manipulation such as deref, assign and new; the latter allocates a memory region and returns its address. Finally, get allows access to a remote resource as if it were local.

Typing rules Since the computations are related to worlds, a term can be given a type only with respect a world. Therefore, the typing judgment has the form "$\Gamma \vdash_{XD} t : A[w]$" which is read "in the context Γ, the term t has type A in the world w". (We will omit the index $_{XD}$ when clear from the context.) The typing contexts *Ctxt* are defined as usual: $\Gamma ::= \langle\rangle \mid \Gamma, x : A[w] \quad x \notin \mathrm{dom}(\Gamma)$

The typing system is given in Figure 1. Many typing rules are straightforward; we just focus on the most peculiar ones.

For monadic types, beside the standard rules for return and bind, we have rules for references and mobile computation. We have chosen that references can contain only naturals (rules *New, Update, Lookup*), but clearly this can be easily generalized. A computation at a given world can be "moved" to another only by means of the get primitive, which can be seen as a "remote procedure call". However, not all computation types can be moved safely; in particular, a reference is meaningful only with respect to the world it was generated from. Therefore, following [7], we define the class of *mobile types*. Intuitively, a type is mobile if its meaning does not depend on the world where it is interpreted. This is formalized by an auxiliary judgment Mobile over types, defined as follows:

$$
\frac{b \in \{\mathsf{Unit}, \mathsf{Nat}, \mathsf{Bool}\}}{\mathsf{Mobile}\ b} \qquad \frac{}{\mathsf{Mobile}\ A@w} \qquad \frac{\mathsf{Mobile}\ A \quad \mathsf{Mobile}\ B}{\mathsf{Mobile}\ A \times B} \qquad \frac{\mathsf{Mobile}\ A\ Q \in \{\forall, \exists\}}{\mathsf{Mobile}\ Qw.A}
$$

$$\frac{\Gamma, x : A[w] \vdash M : B[w]}{\Gamma \vdash \lambda x.M : A \to B[w]}\ Lam \qquad \frac{\Gamma \vdash M : A \to B[w] \quad \Gamma \vdash N : A[w]}{\Gamma \vdash (M\ N) :\ B[w]}\ App$$

$$\frac{\Gamma \vdash M : A[w] \quad \Gamma \vdash N : B[w]}{\Gamma \vdash (M, N) : A \times B[w]}\ Pair \qquad \frac{\Gamma \vdash M : A \times B[w]}{\Gamma \vdash \pi_1 M : A[w]}\ Proj_1 \qquad \frac{\Gamma \vdash M : A \times B[w]}{\Gamma \vdash \pi_2 M : B[w]}\ Proj_2$$

$$\frac{\Gamma \vdash M :\ A[w'] \quad w \text{ fresh in } \Gamma}{\Gamma \vdash \Lambda w.M : \forall w.A[w']}\ Box \qquad \frac{\Gamma \vdash M : \forall w.A[w]}{\Gamma \vdash \mathsf{unbox}\ w' M : A[w]}\ Unbox$$

$$\frac{\vdash \Gamma \quad x : A[w] \in \Gamma}{\Gamma \vdash x : A[w]}\ Var \qquad \frac{\Gamma \vdash M : A[w]}{\Gamma \vdash \mathsf{some}\ w\ M : \exists w.A[w]}\ Some$$

$$\frac{\Gamma \vdash M : \exists u.A[w] \quad \Gamma, x : A\{z/u\}[w] \vdash N : C[w'] \quad z \text{ fresh in } \Gamma}{\Gamma \vdash \mathsf{letd}\ (z, x) = M\ \mathsf{in}\ N : C[w']}\ LetD$$

$$\frac{\Gamma \vdash M : A[w]}{\Gamma \vdash \mathsf{hold}\ M : A@w[w']}\ Hold \qquad \frac{\Gamma \vdash M : A@z[w] \quad \Gamma, x : A[z] \vdash N : C[w]}{\Gamma \vdash \mathsf{leta}\ x = M\ \mathsf{in}\ N : C[w]}\ LetA$$

$$\frac{\Gamma \vdash M : A[w]}{\Gamma \vdash \mathsf{return}\ M : \bigcirc A[w]}\ Ret \qquad \frac{\Gamma \vdash M : \bigcirc A[w] \quad \Gamma \vdash N : A \to \bigcirc B[w]}{\Gamma \vdash \mathsf{bind}\ M\ N : \bigcirc B[w]}\ Bind$$

$$\frac{\mathsf{Mobile}\ A \quad \Gamma \vdash M : \bigcirc A[w']}{\Gamma \vdash \mathsf{get}\ w'\ M : \bigcirc A[w]}\ Get \qquad \frac{\Gamma \vdash M : \mathsf{Nat}[w] \quad \Gamma \vdash N : \mathsf{Ref} \to \bigcirc A[w]}{\Gamma \vdash \mathsf{new}\ M\ N : \bigcirc A[w]}\ New$$

$$\frac{\Gamma \vdash M : \mathsf{Ref}\ [w] \quad \Gamma \vdash N : \mathsf{Nat} \to \bigcirc A}{\Gamma \vdash \mathsf{lookup}\ M\ N : \bigcirc A[w]}\ Lookup$$

$$\frac{\Gamma \vdash T1 : \mathsf{Ref}\ [w] \quad \Gamma \vdash T2 : \mathsf{Nat}[w] \quad \Gamma \vdash T3 : \bigcirc A[w]}{\Gamma \vdash \mathsf{update}\ T1\ T2\ T3 : \bigcirc A[w]}\ Update$$

Fig. 1. Typing system for λ_{XD}.

A computation returning Nat can be accessed remotely, even if it has some side effect on the remote world, like e.g., get w' ((new 0 (λx.lookup x (λy.return y))) (which has type $\bigcirc$Nat at any world w). On the other hand, a computation of type $\bigcirc$Ref cannot be executed remotely. Still, a reference becomes mobile if we indicate which world it comes from, using the hold constructor; e.g., the term get w' ((new 0(λx.return (hold x))) has type $\bigcirc$(Ref@w') at any world w.

4.2 Formalization of λ_{XD} in Coq

The next step is to formalize the object type theory in the Calculus of Inductive Constructions, by means of a "deep encoding" of types and terms.

Types are represented by an inductively defined `Set`. The definition is standard, using the "weak HOAS" technique for representing binders $\forall$ and $\exists$ [3,2]:

```
Hypothesis world : Set.
Inductive typ : Set :=
  typ_unit : typ
| typ_nat : typ
| typ_bool : typ
| typ_pair : typ -> typ -> typ
| typ_fun: typ -> typ -> typ
| typ_forall : (world -> typ) -> typ
| typ_exists : (world -> typ) -> typ
| typ_at : typ -> world -> typ
| typ_monad : typ -> typ
| typ_ref : typ.
```

As an example, $\forall w.A@w$ is represented as `(typ_forall (fun w => (typ_at A w))`. Notice that `world` is not defined inductively (otherwise we would get exotic terms), the only terms inhabiting `world` are variables and possibly constants.

The encoding of terms deals with two kinds of variables and binders. Since we will define a typing judgment with explicit typing contexts, we use de Bruijn indexes for representing first-class variables x; instead, variables w ranging over worlds are represented via weak HOAS, as shown in the following definition:

```
Inductive location : Set := loc : world -> nat -> location.
Inductive term : Set :=
  term_var : nat -> term              | term_pair : term -> term -> term
| term_tt : term                      | term_prj1 : term -> term
| term_location : location -> term    | term_prj2 : term -> term
| term_o : term                       | term_app : term -> term -> term
| term_s : term -> term                | term_lam : term -> term
| term_return : term -> term           | term_hold : term -> term
| term_get : world -> term -> term     | term_leta : term -> term -> term
| term_bind : term -> term -> term     | term_letd : term -> (world -> term)
| term_new : term -> term -> term                   -> term
| term_update : term -> term ->       | term_some : world -> term -> term
                 term -> term         | term_box : (world -> term) -> term
| term_lookup : term -> term -> term | term_unbox : world -> term -> term.
```

Thus, `(typ_fun (typ_var 0))` is the usual "de Bruijn-style" encoding of $\lambda x.x$, while `(typ_box fun w => (term_get w M))` is the encoding of $\Lambda w.(\mathsf{get}\ w\ 0)$ in "weak HOAS style". The constructor letd is represented by `term_letd` using both techniques at once: weak HOAS for the world binding and de Bruijn indexes for the term binding. This mixed approach has many advantages: first, we do not need an additional type to represent term variables, secondly, the correctness is easier to prove. Other language constructors are self-explaining.

Finally the typing judgment $\vdash_{XD}$ is represented by an inductive type as follows (for lack of space, we show only the most complex rule, i.e., that for letd)

```
Inductive typing: env -> term -> typ -> world -> Set := ...
| typing_letd: forall E t1 t2 A C w w',
  E |= t1 ~: typ_exists A [ w' ] ->
   (forall z, E & A z ~ w' |= t2 z ~: C [ w ]) ->
    E |= term_letd t1 t2 ~: C [ w ]
```

where env is the datatype of typing contexts, in de Bruijn style:

```
Inductive env:Set := env_empty:env | env_cons:env -> typ -> world -> env.
```

Intuitively, the term `E |= t ~: A [ w ]` represents the typing judgment $\Gamma \vdash_{XD} M : A[w]$. More formally, we can define the obvious encoding functions $\epsilon_{Terms} : \mathit{Terms} \to \mathtt{term}$, $\epsilon_{Types} : \mathit{Types} \to \mathtt{typ}$, $\epsilon_{Ctxt} : \mathit{Ctxt} \to \mathtt{env}$, such that the following holds (where we omit indexes for readability):

Proposition 1. *For all type A, term M, world w and context Γ, if the worlds appearing free in A, M, Γ, w are $w_1, \ldots, w_n$, then $\Gamma \vdash_{XD} M : A[w] \iff$* `w1:world,...,wn:world` $\vdash$ `_ :` $\epsilon(\Gamma)$ `|=` $\epsilon(M)$`~:` $\epsilon(A)$ `[ w ]`.

This result can be proved by induction on the derivation of $\Gamma \vdash_{XD} M : A[w]$ ($\Rightarrow$) and on the derivation of $\epsilon(\Gamma)$`|=`$\epsilon(M)$ `~:` $\epsilon(A)$ `[ w ]` ($\Leftarrow$). In virtue of this result, in the following we will use the "mathematical" notation for types, terms and typing judgments (i.e., $\Gamma \vdash M : A[w]$) in place of their Coq counterpart.

4.3 Translation of λ_{XD} into CIC `Sets`

In this section we give the translation of types and well-typed terms of the object type theory λ_{XD}, into sets and terms of the Calculus of Inductive Construction, respectively, using the IO monad. This translation can be seen as a way for providing a *shallow* encoding of λ_{XD} in Coq.

Interpretation of object types Object types are interpreted in worlds by a function $\cdot$`<·>` : `typ -> world -> Set`, inductively defined on its first argument:

$$\begin{array}{rlrl}
\mathsf{Unit}\texttt{<w>} &= \texttt{unit} & A \times B\texttt{<w>} &= A\texttt{<w> * } B\texttt{<w>} \\
\mathsf{Nat}\texttt{<w>} &= \texttt{nat} & A \to B\texttt{<w>} &= A\texttt{<w> -> } B\texttt{<w>} \\
\mathsf{Bool}\texttt{<w>} &= \texttt{bool} & \forall w'.A\texttt{<w>} &= \texttt{forall w', (}A\ w'\texttt{) <w>} \\
\mathsf{Ref}\texttt{<w>} &= \texttt{ref } w & \exists w'.A\texttt{<w>} &= \{\ \texttt{w' : world \& (}A\ w'\texttt{) <w>}\ \} \\
A@w'\texttt{<w>} &= A\texttt{<w'>} & \bigcirc A\texttt{<w>} &= \texttt{IO w (}A\texttt{<w>)}
\end{array}$$

Basic types are translated into native Coq types, except for references which are mapped to a specific data type, as explained below. Most constructors are interpreted straightforwardly as well. In particular, note that $\exists w'.A$ is interpreted as a Σ-type, whose elements are pairs `(z,M)` such that `M` has type $(A\ z)$`<w>`. The type $A@w'$ is translated simply by changing the world in which A is interpreted. Finally, the lax type $\bigcirc A$ is translated using the type monad `IO w : Set -> Set`, parametric over worlds, defined in Section 2.

Now we can extend the intepretation of types to typing judgments. Loosely speaking, a judgment $\Gamma \vdash t : A[w]$ should be interpreted as a term of type $\Gamma \to A[w]$. More precisely, this `Set` is obtained by a Coq function $[\![\ ]\!]$: `env -> typ -> world -> Set`, defined recursively over the first argument, such that:

$$[\![(A_1[w_1], ..., A[w_n]), A[w]]\!] = A_1\texttt{<w}_1\texttt{> * ... * }A_n\texttt{<w}_n\texttt{> ->}A\texttt{<w>}$$

Interpretation of object terms Once the type theory has been interpreted as `Set`s, we can give the interpretation of λ_{XD} terms as functions among these sets. Due to lack of space we do not describe in detail this translation, as it is as expected. Most constructors are immediately mapped to their semantic counterparts, e.g., pair is mapped to `pair`, some to `exist`, λ to abstraction, etc. Monadic constructors like `term_return`, `term_get`, ..., are mapped to the corresponding constructors of the IO monad. For more details, we refer to the Coq code.

Soundness Now, we aim to prove that if a λ_{XD} type A is inhabited by a term t, then also the interpretation of A must be inhabited—and the corresponding inhabitant is obtained by translating the object term t. Before stating and proving this result, we need a technical lemma about mobile types:

Lemma 1 (Mobility). *For all $A \in$ Type, if* Mobile *A, then for all w, $w' \in$* `World`*, A* `<w>` $= A$ `<w'>`*.* `[Coq proof]`

This result means that "mobile" types can be translated from one world to another. The Mobile assumption is needed because there are types whose interpretation cannot be moved from one world to another (e.g., Ref). However this "remote access" property is needed only in the case of the get constructor, and its typing rule requires the type to be mobile (Figure 1), so Lemma 1 applies.

Theorem 1 (Soundness). *Let $\Gamma \in Ctxt$, $t \in Term$, $A \in Type$, $w \in World$, let $\{w_1, \ldots, w_n\}$ be all free worlds appearing in Γ, A, t, w. Then, if $\Gamma \vdash_{XD} t : A[w]$ then there exists a term $[\![t]\!]$ such that* `w1:world,...,w2:world` $\vdash [\![t]\!] : [\![\Gamma, A[w]]\!]$. `[Coq proof]`

Proof. (sketch) The proof is by induction on the derivation of $\Gamma \vdash_{XD} t : A[w]$. Most cases are easy; in particular, constructors of monadic types are translated using the constructors defined above.

Let us focus on the case of get, where we move from $\bigcirc A[w']$ to $\bigcirc A[w]$ with the rule *Get*. In the translation we have to build a term of type $\bigcirc A$ `<w>`, i.e., `IO w (`A `<w>)`, from a term in `IO w' (`A `<w'>)` given by inductive hypothesis. In fact, this type is equal to `IO w (`A `<w'>)`, because A is Mobile and by Lemma 1. Then, using `IOget` we can replace `w'` with `w` as required.

For the remaining cases, we refer the reader to the Coq script. □

It is worth noticing that this theorem is stated in Coq as follows

```
Theorem soundness:  forall (t : term),
    forall (w: world), forall (A : typ), forall (E : env),
      E |= t ~: A [ w ] -> Interp E A w.
```

and its proof is precisely the encoding function from λ_{XD} well-typed terms to their semantic interpretations in Coq, i.e., functions among `Set`s.

Remark 2. In this section, we have encoded separately the type of terms and the typing judgment. Another possibility is to define an inductive type of *implicitly typed* terms, representing both the syntax and the typing system, as in [7]:

```
Inductive term : env -> typ -> world -> Set :=
  term_var : nat -> term
| term_o : forall G w, (term G typ_nat w)
| term_s: forall G w, (term G typ_nat w) -> (term G typ_nat w)
| term_fun : forall A B G w, (term G::(A,w) B w) ->
                          (term G (typ_fun A B) w)
| term_app : forall A B G w, (term G (typ_fun A B) w) ->
                          (term G A w) -> (term G B w)
| ...
```

In this way, terms inhabiting `(term G A w)` are automatically well typed. Also the soundness statement would be simplified accordingly:

```
Theorem soundness: forall G A w, (term G A w) -> (Interp G A w).
```

Although this approach may simplify a bit the technical development in Coq, we have preferred to keep separated terms and type system because it is sticking to the original definition "on the paper" (i.e., that in Section 4.1), and hence easier to adapt to other languages and type systems.

5 Example: synthesis of remote read/write

In this section we describe an example application of our framework, showing how to give the specification of two distributed functions which have to satisfy together some property. In particular, we want to show that, if we store a given value in some remote location created on the fly and afterwards we read from the same address, we will find exactly the same value and no error can arise. From the proof of this property, we will extract the corresponding Erlang code, yielding two functions, `remoteread` and `remotewrite`, which behave as required.

We begin with giving the type of the `remotewrite` function which takes the value at the world `w1` and gives a computation at `w1` which produces a location at the world `w2`.

```
Lemma remotewrite: forall w1 w2:world,
            (typ_nat @ w1 ==> (0 (typ_ref@w2))) < w1 >.
```

Notice that this specification does not guarantee that the function will not run without errors, since the monad allows also for faulty computations.

The `remoteread` function takes the reference to the location and gives a computation which returns the value contained in the location. Notice the world of the location is different from the world of the resulting computation:

```
Lemma remoteread: forall w1 w2, typ_ref@w2 ==> (0 typ_nat) < w1 >.
```

Clearly, these two specifications do not impose anything about the combined behaviour of these two functions—they just declare their types. In order to achieve the correct specification, we have to state and prove the following lemma, declaring that there exists a computation and a function which behave correctly:

```
Lemma update: forall (w w': world) (value : typ_nat < w >),
    {o : (0 (typ_ref@w') < w >) *
     ((typ_ref@w') ==> 0 typ_nat < w >) |
      forall s, getvalue ((IObind (fst o) (fun l => (snd o) l)) s)
                 = Some value}.
```

This lemma can be proved using the properties of the monad operators. Then, executing Coq's `Extraction` command, we obtain an Haskell code, which can be translated to Erlang by our back-end compiler. In the end, we obtain the following distributed code:

```
remotewrite (W1, W2, Value)->
  spawn (element (2, W1),
         update2,
         dispatcher,
         [update2,
          (fun () -> (fun (Address)-> Address end)
                     (spawn (element (2, W2),
                     update2,
                     location, [Value])) end),
          W2, {self (), node ()}]),
  receive {result, Z} -> Z end .
```

```
remoteread (W1, W2, Address)->
  spawn (element (2, W1),
         update2,
         dispatcher,
         [update2,
          (fun () -> Address ! {get, {self (), node ()}},
                     receive {result, X0} ->
                        (fun (H)-> H end)(X0)
                     end end),
          W2, {self (), node ()}]),
  receive {result, Z} -> Z end.
```

The function `remotewrite` spawns a process at `W1` with final destination `W2`, which creates a new location with `Value` as initial value and waits for the address of that location. On the other hand, `remoteread` spawns a process to `W1` with final target `W2`, which sends to the `Address` location a request for the value and waits for the response. The target process computes and sends back the result of the computation, which is received by the final `receive`.

Clearly, these functions can execute only when they are called by some other program. we can define the `main/0` function, which can be seen as the "orchestration" part for executing the distributed code. As an example, let us consider the following implementation where we declare the remote world `Pub` and the `Dispatcher`, then `remotewrite` will create a new location and put in it the value 254. `remoteread` is then called to read the same value from the same address:

```
main() -> Pub = {pub, 'pub@<IP>'},
          Dispatcher = {disp, 'disp@<IP>'},
          Val = 254,
          Address = remotewrite(Dispatcher, Pub, Val),
          Value = remoteread(Dispatcher, Pub, Address).
```

6 Conclusions

In this work we have presented a methodology for the synthesis of distributed, mobile programs in Erlang, through the extraction facility offered by Coq. First, we have defined monadic types in Coq, covering the computational features we are dealing with (i.e., side-effects and distributed computation). These monadic types can be used in the specification of Haskell programs, which can be obtained by extraction from the proofs of these specifications. These programs contain monadic constructors for distributed imperative computations, which are exploited by a "post-extraction" Haskell-to-Erlang compiler, which generates the requested distributed mobile program. Moreover, in order to simplify the burden of programming in Coq, we have defined λ_{XD}, a monadic type theory for distributed computations similar to Licata and Harper's HL5, which can be seen as a *front-end* programming language. In fact, this type theory has been given a formal interpretation within the Calculus of Inductive Constructions, which allows a λ_{XD} type to be converted into a CIC specification and a λ_{XD} program into a CIC proof. Using the back-end compiler above, these proofs can be turned into runnable Erlang programs.

Several directions for future work stem from the present one. First, we can extend the language to consider also worlds as first-class objects. This would allow computations to take and returns "worlds", so that a program can dynamically choose the world where the execution should be performed.

Although we have considered distributed computations with references, our approach can be ported to other computational aspects, possibly implemented in other target languages. The monad `IO` has to be extended with new (or different) constructors (possibly in a quite different target language), the post-extraction back-end compiler should be extended to cover these new constructors, and the front-end type theory must be adapted as needed.

However, the most important future work is to prove that the post-extraction compiler, i.e., the translation from Haskell to Erlang, is correct. To this end, we could follow the approach of the CompCert project described by Leroy [4,5]. This can be achieved by giving in Coq a formal semantics to the fragments of Haskell and Erlang that the back-end compiler targets. In particular, the crux of the correctness proof is proving that the Erlang implementations of mobility, plus state effects as threads, are correct with respect to the "reference implementation" of the corresponding monad operators.

Acknowledgments. The authors are grateful to the anonymous reviewers for many useful and important remarks, which allowed to quite improve the paper.

References

1. J. Armstrong. Erlang - a survey of the language and its industrial applications. In *Proc. INAP'96*, 1996.
2. F. Honsell and M. Miculan. A natural deduction approach to dynamic logics. In S. Berardi and M. Coppo, editors, *Proc. of TYPES'95*, volume 1158 of *Lecture Notes in Computer Science*, pages 165–182, Turin, 1995. Springer-Verlag, 1996.
3. F. Honsell, M. Miculan, and I. Scagnetto. π-calculus in (co)inductive type theory. *Theoretical Computer Science*, 253(2):239–285, 2001.
4. X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. POPL*, pages 42–54. ACM, 2006.
5. X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.
6. 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.
7. D. R. Licata and R. Harper. A monadic formalization of ML5. In K. Crary and M. Miculan, editors, *Proc. LFMTP*, volume 34 of *EPTCS*, pages 69–83, 2010.
8. T. V. Murphy, K. Crary, and R. Harper. Type-safe distributed programming with ML5. In G. Barthe and C. Fournet, editors, *Proc. TGC*, volume 4912 of *Lecture Notes in Computer Science*, pages 108–123. Springer, 2007.
9. C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.
10. 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.
11. R. Virding, C. Wikström, and M. Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., 1996.