Implementation and Verification of Modular Effectful Systems in Coq using FreeSpec

Yann Régis-Gianas
(IRIF, Univ. Paris Diderot, Inria) – yrg@irif.fr

with Thomas Letan (ANSSI), Vincent Tourneur (Inria, IRIF), Pierre Chifflier (ANSSI) and Guillaume Hiet (CentraleSupélec, Inria)

2019-09-03 – PPS days
A classic IRC joke about Coq programmers

someone> Could you write a certified compiler for me?
coq-der> My pleasure!
A classic IRC joke about Coq programmers

someone> Could you write a certified compiler for me?
cq-der> My pleasure!

someone> Could you write a hello world program for me?
cq-der> No.
-!- cq-der [~x1@166.37.73.42] has left #coq [Crying]
A classic IRC joke about Coq programmers

someone> Could you write a certified compiler for me?
coq-der> My pleasure!

someone> Could you write a hello world program for me?
coq-der> No.
-!- coq-der [~x1@166.37.73.42] has left #coq [Crying]

This joke does not work anymore.
Let’s make Gallina a general purpose programming language!
This talk

FreeSpec is a Coq library and plugin to develop effectful systems in Gallina

What are the design choices of FreeSpec?

- How to write **effectful programs**?
- How to **build large systems** by composition of effectful components?
- How to **specify them**?
- How to **reason about them**?
- What are the **limitations** of FreeSpec?
How to write **effectful programs** in Gallina?
A rich design space of representations for effectful programs

One Monad to Prove Them All
Sandra Dylus, Jan Christiansen, and Finn Teegen

Programming and Reasoning with Algebraic Effects and Dependent Types
Edwin C. Brady
A rich design space of representations for effecful programs

One Monad to Prove Them All
Sandra Dylus\textsuperscript{a}, Jan Christiansen\textsuperscript{b}, and Finn Teegen\textsuperscript{a}

Programming and Reasoning with Algebraic Effects and Dependent Types
Edwin C. Brady

FreeSpec focuses on
Modularity of implementations, specifications and proofs.
Interface: a collection of effects with heterogeneous types of answers.
Effectful operations as interfaces

**Interface**: a collection of effects with heterogeneous types of answers.

```
Definition Interface := Type -> Type.
```
Effectful operations as interfaces

**Interface**: a collection of effects with heterogeneous types of answers.

```
Definition Interface := Type -> Type.
```

**Example**: Interface for basic terminal interaction.

```
Inductive i : Type -> Type :=
| Scan : i string
| Echo : string -> i unit.
```
Effectful operations as interfaces

**Interface**: a collection of effects with heterogeneous types of answers.

```plaintext
Definition Interface := Type -> Type.
```

Example: **Interface for file manipulation.**

```plaintext
Inductive i: Type -> Type :=
  | Stat: string -> i stats
  | Open: mode -> options -> string -> i fd
  | OpenDir: string -> i fd
  | FStat: fd -> i stats
  | GetSize: fd -> i N
  | Read: N -> fd -> i string
  | ReadDir: fd -> i string
  | Write: string -> fd -> i unit
  | Seek: seekRef -> fd -> fd -> i unit
  | Close: fd -> i unit
  | CloseDir: fd -> i unit.
```
A **program** either purely computes or it interacts with its environment.

```plaintext
Inductive Program (I: Interface) (A: Type) :=
  | Pure (a: A) : Program I A
  | Request {B: Type} (e: I B) (f: B -> Program I A) : Program I A.
```
Programs as inhabitants of a free monad

A **program** either purely computes or it interacts with its environment.

``` Ocaml
Inductive Program (I : Interface) (A : Type) :=
| Pure (a : A) : Program I A
| Request {B : Type} (e : I B) (f : B -> Program I A) : Program I A.
```

**Example**

``` Ocaml
Definition hello : Program Console.i unit :=
  Request Console.Scan (fun name =>
  Request (Console.Echo ("Hello " ++ name)) (fun _ =>
  Pure tt)).
```
A program either purely computes or it interacts with its environment.

```
Inductive Program (I: Interface) (A: Type) :=
  | Pure (a: A) : Program I A
  | Request {B: Type} (e: I B) (f: B -> Program I A) : Program I A.
```

Example

```
Definition hello {ix} ~(Use Console.i ix) : Program ix unit :=
    name <- scan;
    echo ("Hello " ++ name).
```
Let us have a look at the implementation of coqar, a version of ar implemented using FreeSpec.¹

¹Work-in-progress with Vincent Tourneur and Thomas Letan.
An archive produced by **ar**

An **ar** file is a textual representation of the concatenation of several files:

```
!<arch>
test1.txt/ 0 0 0 644 14 ~
coqar example
test2.txt/ 0 0 0 644 8 ~
bonjour
```
Definition create {ix} `{Use FileSystem.i ix}
  (files : list string) (output : string)
: Program ix unit :=
  fd <- open WriteOnly MayCreateTruncate output;
  write_header fd;;
  insert_files fd files;;
  close fd.
Fixpoint insert_files {ix} `{Use FileSystem.i ix}
  (fd : fd) (files : list string)
: Program ix unit :=
  match files with
  | file :: l =>
    write_entry fd file;;
    insert_files fd l
  | _ => pure tt
  end.
Definition write_entry \{\text{ix}\} \{\text{Use FileSystem.i ix}\}
  (\text{fd : fd}) (\text{name : string})
: Program \text{ix} \ \text{unit} :=
  \text{fd2} \leftarrow \text{open Readonly DontCreate name};
  \text{size} \leftarrow \text{getSize fd2};
  \text{content} \leftarrow \text{read size fd2};
  \text{close fd2};
  \text{write (gen_header name 644 size) fd};
  \text{write new_line fd};
  \text{write content fd}.
Definition write_entry {ix} `{Use FileSystem.i ix}`
  (fd : fd) (name : string)
  : Program ix unit :=
  fd2 <- open ReadOnly DontCreate name;
  size <- getSize fd2;
  content <- read size fd2;
  close fd2;;
  write (gen_header name 644 size) fd;;
  write new_line fd;;
  write content fd.
Coinductive interpretation of programs

▶ As is, a term of type `Program I A` is like an empty shell.
▶ We must give a meaning to each effectful operation.
▶ This interpretation is potentially modified after each effect.
▶ Hence, this updated interpretation must be returned.

CoInductive Semantics : Type :=
| handler (f: forall {A: Type}, I A -> Result A): Semantics

with Result: Type -> Type :=
| mkRes {A}: A -> Semantics -> Result A.

▶ This idea comes from the Haskell `operational` package.
▶ The “stream-like” type of `handler` imposes a coinductive definition.
Semantics are easily equipped with an evolving state.

```ocaml
Definition PS {I : Interface} (State : Type) :=
  forall (A: Type), State -> I A -> (A * State).

CoFixpoint mkSemantics {I : Interface} {State : Type}
  (ps : PS State) (s : State) : Semantics I :=
  handler (fun (A: Type) (e: I A) =>
    mkRes (fst (ps A s e)) (mkSemantics ps (snd (ps A s e))))).
```
A strength of FreeSpec: Program evaluation

Program evaluation is defined by induction.

```plaintext
Fixpoint runProgram
  {I: Interface}
  {A: Type}
  (sig: Semantics I)
  (p: Program I A)
  : Result I A :=
  match p with
  | Pure a =>
    mkResult a sig
  | Request e f =>
    let res := handle sig e in
    runProgram (Semantics.next res) (f (Semantics.result res))
  end.
```
How to realize such an **effect handler**?
How to realize such an **effect handler**?

By simulation – By extraction – By interpretation – By delegation
Realization of semantics: by simulation

A Coq function can act as a denotation for any semantics.

```
Inductive NatStack : Type -> Type :=
| Push (x: nat) : NatStack unit |
| Pop : NatStack (option nat).
```

```
Definition nat_stack_semantics : Semantics NatStack :=
  mkSemantics (fun (A: Type) (l: list nat) (e: NatStack A) =>
    match e with
    | Push x => (tt, x :: l)
    | Pop => match l with
       | x :: r => (Some x, r)
       | _ => (None, nil)
    end
  end) nil.
```
Realization of semantics: by extraction

- Program evaluators extract to straightforward OCaml code.
- Impure functions serve as effective semantics for effectful operations.

```ocaml
Fixpoint pipe {ix} ~{Use Console.i ix} n : Program ix unit :=
  match n with
  | O => pure tt
  | S k => Console.scan >>= Console.echo;; pipe k
end.

Axiom ocaml_scan : unit -> string.
Axiom ocaml_echo : string -> unit.

CoFixpoint ocaml_semantics :=
  handler (fun {A} (x : Console.i A) =>
    match x with
    | Console.Scan => Sem.mkRes (ocaml_scan tt) ocaml_semantics
    | Console.Echo s => Sem.mkRes (ocaml_echo s) ocaml_semantics
  end).

Definition run_pipe : unit := evalProgram ocaml_semantics (pipe 10).

Extraction "pipe.ml" run_pipe.
```
Realization of semantics: by interpretation

Or else...

```plaintext
Fixpoint pipe {ix} `{Use Console.i ix} n: Program ix unit :=
    match n with
    | 0 => pure tt
    | S k => Console.scan >>= Console.echo;; pipe k
    end.

Exec (pipe 10).
```
Realization of semantics: by interpretation

What is Exec?

- Exec is a plugin written in OCaml inspired by the M tac interpreter.
- It interprets any Program I A directly (no extraction or compilation).
What is \texttt{Exec}?

- \texttt{Exec} is a plugin written in OCaml inspired by the Mtac interpreter.
- It interprets any \texttt{Program I A} directly (no extraction or compilation).

Bi-interpreter

Let \( t \) be of type \texttt{Program I A}.

1. Compute \( w \), the weak-head normal form of \( t \).
2. If \( w \) is \texttt{Pure u}, returns \( u \).
3. If \( w \) is \texttt{Request e f}, pass the normal form \( c \) of \( e \) to an effect handler \( g \) written in OCaml, go back to (1) with \( t = f(g \ c) \).
How to compose large systems made of effectful components?
Realization of semantics: by delegation

Modularity of implementations:

Your handler is someone else's component (and conversely).
Realization of semantics: by delegation

Modularity of implementations:

Your handler is someone else’s component (and conversely).

1 (* I is the interface implemented by the component. *)
2 (* J is the dependency of the component. *)
3 Definition Component (I: Interface) (S: Type) (J: Interface) :=
4     forall (A: Type), I A -> S -> Program J A * S.
Realization of semantics: by delegation

Modularity of implementations:

Your handler is someone else’s component (and conversely).

```
(* I is the interface implemented by the component. *)
(* J is the dependency of the component. *)
Definition Component (I: Interface) (S: Type) (J: Interface) :=
    forall (A: Type), I A -> S -> Program J A * S.
```

```
Definition ComponentSemantics {I J: Interface} {S: Type}
    (c: Component I S J) (s: S) (sig: Semantics J)
: Semantics I :=
    mkSemantics (fun {A: Type} (s': (S * Semantics J)) (e: I A) =>
        (* ... Implement interface I in terms of interface J ... *)
```
Visual examples
Composing interfaces

Modularity of interfaces:

\[
\text{Inductive} \text{ ComposedInterface } (I \ J : \text{Interface}) \ (A: \text{Type}) : \text{Type} := \\
\text{InL} (e : I \ A) : \text{ComposedInterface} \ I \ J \ A \\
\text{InR} (e : J \ A) : \text{ComposedInterface} \ I \ J \ A.
\]

Infix "<+>" := (ComposedInterface) (at level 50, left associativity).

\[
\text{CoFixpoint} \ mkCompSemantics \ {I J : \text{Interface}} \\
(s_i : \text{Semantics} \ I) (s_j : \text{Semantics} \ J) : \text{Semantics} (I <+> J) \\
:= (*) \ldots (*)
\]
How to **specify** large systems made of effectful components?
Since components are stateful, specifications must refer to a state.

To preserve encapsulation, we use a notion of **abstract state**.

An interface specification over abstract state type $W$ is:

```ocaml
Record InterfaceSpecification (W : Type) (I : Interface) : Type :=
{
  (* An interpreter for effects over abstract states. *)

  (* The effects that are compatible with a given abstract state. *)
  allowed_operation : W -> forall A, I A -> Prop;

  (* The effect answers that can arise in a given abstract state. *)
}.
```
An abstract state for the FileSystem interface

Definition partial (A B : Type) := A -> option B.

Definition fdContent := partial fd ascii.

Record fdState : Type := MkFdState
{
    mode : FileSystem.mode;
    kind : option FileSystem.fileKind;
    size : option nat;
    pos : option nat;
    content : fdContent;
}.

Definition state := partial fd fdState.

This abstract state represents partial information about the file system.
An abstract step function for the FileSystem interface

Here is the case for `Open`:

```plaintext
abstract_step (FileSystem.Open m o str) fd s :=
    let size :=
        match o with
        | DontCreateTruncate | MayCreateTruncate | MustCreate => Some 0
        | _ => None
    end
    in
    setFun s fd {|
        mode := m;
        kind := None;
        size := size;
        pos := Some 0
        content := const None;
    |};

(* ... *)
```

This function refines the static knowledge about the file system.
What is a good abstraction specification for FileSystem?

Designing specifications is hard

- This specification of the FileSystem interface is simple.
- Simplicity makes the reviewing of proof assumptions tractable.
- Yet, the path is short from “simple” to “simplistic”.

FreeSpec’s philosophy

- FreeSpec allows multiple specifications for a single interface.
- One can check that these specifications are not contradictory.
Compositionality of abstract specifications

Modularity of specifications

Theorem compose_specifications \{W_I W_J: Type\} \{I J: Interface\}
(c_i: Specification W_I I)
(c_j: Specification W_J J)

Theorem expand_specification_left \{W: Type\} \{I: Interface\}
(c: Specification W I)
(J: Interface)

Theorem expand_specification_right \{W: Type\} \{J: Interface\}
(c: Specification W J)
(I: Interface)
How to **prove** large systems made of effectful components?
Two key notions: Compliance and Respectfulness

Semantics compliance
The coinductive predicate \( \text{semantics} \models s[w] \) means
“For every operation of the \text{semantics} interface which is allowed by specification \( s \) in abstract state \( w \), the \text{semantics} only produces expected answers, and stays compliant on the next abstract step.”

Program respectfulness
The inductive predicate \( p \triangleright s[w] \) means
“The program \( p \) only performs operations that are allowed by the specification \( s \) in abstract state \( w \) and for every valid answers, the continuation of the program stays respectful on the next abstract step.”
Component correctness

The two sides of the component verification problem

To be correct, a Component I S J:

- must be respectful of J, and
- must fulfill the requirements of I.

Difficulty

- The abstract states for I and J are interdependent.
- Their relationship is crucial for the correctness proof of the component.
- Hence the need for a synchronization predicate:

```plaintext
Definition sync_pred (W_I W_J: Type) (S: Type) :=
W_I -> S -> W_J -> Prop.
```
Component respectfulness

Informally

A component is **respectful** if it uses its dependencies in a respectful way when it is itself used with respect.
Informally

A component is **respectful** if it uses its dependencies in a respectful way when it is itself used with respect.

```
Definition respectful_component
  {W_I W_J: Type}
  {I J: Interface}
  {S: Type}
  (component: Component I S J)
  (master: Specification W_I I)
  (slave: Specification W_J J)
  (sync: sync_pred W_I W_J S) :=
  forall (w_i: W_I) (s: S) (w_j: W_J) {A: Type} (e: I A),
      sync w_i s w_j
  -> allowed_operation master e w_i
  -> (component A e s) |> slave[w_j].
```
Reasoning principle for Component Compliance

Informally

If the synchronization predicate stays valid during component’s execution and is strong enough to imply the expectations about effects’ answers, then the component will be compliant when composed with any compliant semantics.
Reasoning principle for Component Compliance

Informally

If the synchronization predicate stays valid during component’s execution and is strong enough to imply the expectations about effects’ answers, then the component will be compliant when composed with any compliant semantics.

```
Theorem compliant_component {W_I W_J: Type} {I J: Interface} {S: Type}
  (component: Component I S J)
  (master: Specification W_I I)
  (slave: Specification W_J J)
  (sync: sync_pred W_I W_J S)
  (Hsyncpres: sync_preservation component master slave sync)
  (Hsyncp: sync_postcondition component master slave sync)
  (Hrespectful: respectful_component component master slave sync)
  : forall (w_i: W_I)
    (s: S)
    (w_j: W_J)
    (sig: Sem.t J)
    (Hcomp: sig |= slave[w_j]),
    sync w_i s w_j
  -> ComponentSemantics component s sig |= master[w_i].
```
Verification Methodology

Verifying the CPU
Verification Methodology

Verifying the CPU

The CPU executes **software** thanks to a **memory controller**.

![Diagram of CPU and memory controller](image)

\[ I_{CPU} \xrightarrow{CPU} I_{MC} \]
Verification Methodology

Verifying the CPU

We want to prove the CPU complies with an interface specification \((A, X)\).

\[
\begin{array}{c}
\text{Software} \quad I_{CPU} \quad \text{CPU} \quad I_{MC} \quad \text{Memory Controller}
\end{array}
\]

\[
\begin{array}{c}
A \\
\hline
\text{CPU} \quad I_{MC}
\end{array}
\]

\[
\begin{array}{c}
X
\end{array}
\]
Verification Methodology

Verifying the CPU

We assume the software component will be respectful.
Verification Methodology

Verifying the CPU

We want to verify that, according to the CPU, the expectations $X$ holds.

\[
\begin{array}{c}
\text{Software} \xrightarrow{I_{CPU}} \text{CPU} \xrightarrow{I_{MC}} \text{Memory Controller}
\end{array}
\]
Verification Methodology

Verifying the CPU

Rather than relying on the memory controller model, we’d rather identify a sufficient couple \((A', X')\), and abstract away the memory controller.
Verification Methodology

Verifying the CPU

We prove that, because $A$ is assumed and thanks to the CPU model, then the assumptions $A'$ are met.
We assume the memory controller fulfills the expectations $X'$. 

\[
\begin{array}{c}
\text{Software} & \xrightarrow{I_{\text{CPU}}} & \text{CPU} & \xrightarrow{I_{\text{MC}}} & \text{Memory Controller} \\
\end{array}
\]
Verification Methodology

Verifying the CPU

We prove that, because $X'$ and thanks to CPU model, then $X$ is verified.
Verification Methodology

Verifying the CPU

In other words, our CPU model enforces $X$ as long as the software component complies to $A$.
Verification Methodology

Composing Verification Results

We can follow a similar proof scheme with the memory controller model.
Verification Methodology
Composing Verification Results
Verification Methodology

Composing Verification Results

![Diagram showing the flow of signals between components: Software, CPU, Memory Controller, and VGA Controller. Signals are represented as $I_{CPU}$ and $I_{MC}$.]
Verification Methodology

Composing Verification Results

Software $\xrightarrow{I_{\text{CPU}}}$ CPU $\xrightarrow{I_{\text{MC}}}$ Memory Controller

- CPU
- DRAM Controller
- VGA Controller
### Theorems about `coqar`

```coq
Definition ar_id {ix} `{Use FileSystem.i ix}
    (files : list string) (ar : string) : Program ix unit :=
    create files ar;;
    remove_files files;;
    extract ar;

Theorem ar_spec :
    ar_id files_names output |> specs[initial].

Theorem ar_correct (final : state) (r : unit)
    : correct_run specs initial (ar_id file_names output) r final
    -> sameFileContents initial final.
```
What are the limitations of FreeSpec?
Interaction trees\(^2\) are coinductively defined:

\[
\begin{align*}
\text{CoInductive} & \quad \text{itree} \quad (E : \text{Type} \rightarrow \text{Type}) \quad (R : \text{Type}) : \text{Type} := \\
| & \quad \text{Ret} \quad (r : R) \\
| & \quad \text{Tau} \quad (t : \text{itree} \ E \ R) \\
| & \quad \text{Vis} \quad \{A : \text{Type}\} \quad (e : E \ A) \quad (k : A \rightarrow \text{itree} \ E \ R).
\end{align*}
\]

... which allows for the representation of diverging computations.

- We could probably patch \texttt{Program} to match this definition.
- This would probably increase the complexity of \texttt{FreeSpec}.
General recursion

- Interaction trees\(^2\) are coinductively defined:

```ocaml
CoInductive itree (E : Type -> Type) (R : Type) : Type :=
  Ret (r : R)
  Tau (t : itree E R)
  Vis {A : Type} (e : E A) (k : A -> itree E R).
```

... which allows for the representation of diverging computations.

- We could probably patch **Program** to match this definition.

- This would probably increase the complexity of FreeSpec.

- We prefer to keep the terminating-by-default behavior ...

...seeing **divergence as an effect** per se.

\(^2\)DeepSpec
Looping as an effect

```plaintext
Inductive interruption A B :=
| paused : A -> interruption A B
| exited : B -> interruption A B.

Inductive loop I A B : Interface :=
| Loop :
  (* Initial state *) A
  (* Loop body *) -> (A -> Program I (interruption A B))
  -> i I A B (option B).
```

- The type `option B` witnesses the fact that a loop may diverge.
- A semantics for `loop` can accumulate the stream of states.
- Contrary to `ITree`, `FreeSpec` has no support for reasoning about these potentially infinite computations.
- Designing reasoning principles about control operators is future work.
Shared state

Problem
If two components depend on the same component, a diamond appears.
Shared state

Problem
If two components depend on the same component, a diamond appears.
Efficiency

- Exec is a nice convenience but is not tuned for performance.
- Will CertiCoq save us from this inefficiency?
- We need partial evaluation to remove the effect-interpretation layer.
Conclusion
FreeSpec, in a nutshell

**FreeSpec**

- uses free monads to represent effectful programs in Coq;
- is able to run effectful programs using `Exec`;
- specifies components with abstract specifications;
- offers reasoning principles to prove them correct;
- in a modular way.

Have a look at our FM’18 paper and at the released code:

http://www.github.com/ANSSI-FR/FreeSpec
FreeSpec, in a nutshell

FreeSpec

- uses free monads to represent effectful programs in Coq;
- is able to run effectful programs using \texttt{Exec};
- specifies components with abstract specifications;
- offers reasoning principles to prove them correct;
- in a modular way.

Have a look at our FM’18 paper and at the released code:

\url{http://www.github.com/ANSSI-FR/FreeSpec}

Thank you for your attention! Questions?