FreeSpec: Specifying, Verifying and Executing Impure Computations in Coq

Thomas Letan, Yann Régis-Gianas

▶ **To cite this version:**

Thomas Letan, Yann Régis-Gianas. FreeSpec: Specifying, Verifying and Executing Impure Computations in Coq. CPP 2020 - 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, Jan 2020, Nouvelle-Orléans, United States. pp.1-15, 10.1145/3372885.3373812 . hal-02422273

HAL Id: hal-02422273

https://inria.hal.science/hal-02422273

Submitted on 21 Dec 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

FreeSpec: Specifying, Verifying and Executing Impure Computations in Coq

Thomas Letan
Software Security Lab
French Cybersecurity Agency (ANSSI)
France
thomas.letan@ssi.gouv.fr

Yann Régis-Gianas
Université de Paris, IRIF/PPS
France
PiR2, Inria Paris-Rocquencourt
France
yrg@irif.fr

Abstract

FreeSpec is a framework for the Coq theorem prover which allows for specifying and verifying complex systems as hierarchies of components verified both in isolation and in composition. While FreeSpec was originally introduced for reasoning about hardware architectures, in this article we propose a novel iteration of FreeSpec formalism specifically designed to write certified programs and libraries. Then, we present in depth how we use this formalism to verify a static files webserver. We use this opportunity to present FreeSpec proof automation tactics, and to demonstrate how they successfully erase FreeSpec internal definitions to let users focus on the core of their proofs. Finally, we introduce FreeSpec.Exec, a plugin for Coq to seamlessly execute certified programs written with FreeSpec.

• **Theory of computation → Program verification**.

certified programs, certified libraries, Coq, framework, proof automation

ACM Reference Format:
Thomas Letan and Yann Régis-Gianas. 2020. FreeSpec: Specifying, Verifying and Executing Impure Computations in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '20), January 20–21, 2020, New Orleans, LA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3372885.3373812

1 Introduction

In previous work, we have introduced FreeSpec [23], a framework for compositional reasoning using the Coq proof assistant [18], distributed as Free Software under the terms of the GPLv3 license[1]. FreeSpec's initial goals were to model complex systems —*e.g.*, an x86 hardware platform— as hierarchies of components interacting through interfaces. A component exposes an interface, uses interfaces exposed by other components, and makes its internal state evolve each time it is executed. FreeSpec provides reasoning tools to verify each component in isolation and the necessary abstractions to compose these components and their proofs into a correctness proof of the overall system. To achieve this in Coq, FreeSpec models a stateful component as an impure computation encoded in Gallina with a variant of the Free monad introduced in the Haskell `operational` package. In that formalism, an interface `i` is defined using an inductive type where each constructor `P` has a type of the form $\texttt{t}_1$ `->` `..` `->` $\texttt{t}_n$ `->` `i` $\texttt{t}_r$ and denotes a primitive with results of type $\texttt{t}_r$. Since both interfaces and impure computations are modeled using algebraic datatypes, it becomes possible to write Gallina predicates to specify how primitives have to be used on the one hand, and to prove that a given impure computation correctly uses them on the other hand.

FreeSpec has never been intended to be limited to the modeling of interconnected hardware components that together form a hardware architecture. In particular, we suspected that FreeSpec could be well suited to reason about so-called "impure programs" —that is programs with side effects. Indeed, monads have long been used to write such programs in purely functional programming languages, Haskell being the most famous herald of this approach. Once we started experimenting with FreeSpec to write and verify impure programs, we noticed our approach to tackle this new use case was fundamentally different from what we had experienced when modeling hardware specifications. In the latter case, the complete architecture of the modeled systems was already known, in particular in terms of interfaces users. On the contrary, software development often builds up on libraries providing general-purpose, reusable functions, and the first iteration of FreeSpec was poorly equipped to write certified libraries to be used in a large variety of contexts.

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

CPP '20, January 20–21, 2020, New Orleans, LA, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-7097-4/20/01…$15.00
https://doi.org/10.1145/3372885.3373812

[1] FreeSpec source code can be downloaded at https://github.com/ANSSI-FR/FreeSpec.

In this article, we report on our progress in turning FreeSpec into a framework to write certified, "impure" programs and libraries. More precisely, our contribution is threefold.

- We present a new iteration of the FreeSpec formalism which focuses on allowing to write and verify general-purpose, reusable functions, *i.e.*, writing certified libraries. In practice, this means we provide new facilities to compose both impure programs, and their correctness proofs (Section 2).
- We introduce FreeSpec.Exec, a proof-of-concept plugin for the Coq theorem prover which turns `coqc`, i.e., the Coq compiler, into an effectful interpreter for FreeSpec's impure computations (Section 3).
- We illustrate how FreeSpec can be used in practice to prove impure computations. Our framework comes with two key tactics which completely erase the Free monad-related boilerplate of FreeSpec's formalism. To illustrate our rationale, we explain in length a simple use case: we specify a minimal, single-threaded, single-connection web server whose only feature is to serve static files, and we prove this program makes a correct use of the file descriptors it manipulates. More precisely, it does not try to read from or close a file which is not currently open, and it correctly closes any file descriptor it creates (Section 4). The resulting project is called `MiniHTTPServer`, can be executed by FreeSpec.Exec and is distributed under the terms of the GPLv3 license [2].

We conclude this article by discussing FreeSpec w.r.t. related work (Section 5), and by detailing future work (Section 6).

2 FreeSpec Formalism

In this section, we recall the basic concepts on top of which FreeSpec's users can specify and verify impure computations in a modular way. Modularity is achieved thanks to a reasoning framework in line with Floyd-Hoare Logic and the "design by contract" approach by Meyer [27], where impure computations interact with stateful external components through a well-defined interface. Correctness is then considered through these interactions: as long as an impure computation (the caller) uses a given interface "correctly," then the component (the callee) computes "correct" results, which conversely allow the caller to continue to use the interface "correctly." Caller and callee obligations are grouped together in an *interface contract*.

In this paper, we assume that the reader has some basic knowledge of the Coq proof assistant especially of its type class mechanisms [32].

[2] `MiniHTTPServer` source code can be downloaded at https://github.com/ANSSI-FR/coq-MiniHTTPServer.

2.1 Specifying Impure Computations

An interface is a set of interdependent primitives expected to produce a result. In FreeSpec, interfaces are encoded as parameterized types, that is

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

Given `i` an interface and `t` a type, a term of type `i t` identifies a primitive of `i` expected to produce a result of type `t`. As an example, we consider the `STORE s` interface. Primitives of type `STORE s t` allow for manipulating a global and mutable variable of type `s` and produces a value of type `t`. Specifying the `STORE s` interface is straightforward with an inductive type.

```
Inductive STORE (s : Type) : interface :=
| Get : STORE s s
| Put (x : s) : STORE s unit.
```

This definition reads as follows: impure computations manipulating a global variable can either retrieve its current value (hence `Get` answers a value of type `s`), or modify it (hence `Put` expects a term of type `s`).

We emphasize that terms of type `STORE s` *identify* primitives, and do not *implement* them. Similarly, we also encode an impure computation leveraging an interface `i` to compute results of a type `t` with an inductive type `impure i t` defined as follows:

```
Inductive impure (i : interface) (t : Type) :=
| local (x : t) : impure i t
| request_then {u} (p : i u)
    (f : u -> impure i t)
  : impure i t.
```

An impure computation is either the resulting term `x` of a local computation, or the action of requesting the environment to handle a primitive `p`. The second argument of `request_then` is a continuation `f` which expects the result of `p` as its argument.

For instance, we can implement an impure computation which, given a global variable of type `nat`, increments this variable and returns its previous value as follows:

```
Definition get_inc : impure (STORE nat) nat :=
  request_then Get
    (fun x => request_then (Put (S x))
                (fun _ => local x)).
```

2.2 Composite Interfaces and Polymorphism

The `impure` type is a variant of the Free monad, and we can compose impure computations which use the same interface together using the `bind` combinator whose type is

```
  impure i u -> (u -> impure i t) -> impure i t
```

and whose definition is straightforward:

```
Fixpoint impure_bind {i a b}
  (p : impure i a) (f : a -> impure i b)
: impure i b :=
  match p with
  | local x =>
    f x
  | request_then e g =>
    request_then e (fun x => impure_bind (g x) f)
  end.
```

The monad formalism and its `bind` combinator in particular have proven to be an effective abstraction to define and reuse computations, but from our perspective, forcing computations to use the same interface is a significant limitation. In practice, we indeed want to define computations which use more than one interface (*e.g.*, a daemon which serves static files over TCP sockets). In our previous work [23], we were addressing this challenge thanks to a dedicated composition operator for interfaces. In this paper, we introduce a more general approach, based on so-called composite interfaces which allow for defining impure computations over a *polymorphic* composite interface.

We say that a composite interface `ix` provides an interface `i` if there exists a function `inj_p {t} : i t -> ix t`. Conversely, we can determine if a primitive of a composite interface `ix` is forwarded to an interface `i` when we have a function `proj_p {t} : ix t -> option (i t)`. The definitions of `proj_p` and `inj_p` shall obey two laws:

1. Assuming `ix` provides `i`, then given `p` a primitive of `i`, injecting `p` in `ix` then projecting the result back in `i` gives back `p`, i.e., `proj_p (inj_p p) = Some p`
2. Assuming `i` and `j` are two different interfaces, if `ix` provides `i`, then given `p` a primitive of `i`, injecting `p` in `ix`, projecting the resulting primitive in `j` returns None, i.e., `proj_p (i:=j) (inj_p e) = None`

`MayProvide`, `Provide` and `Distinguish` form a dedicated type class hierarchy that implements this mechanism in FreeSpec (Figure 1). These type classes are transparent to FreeSpec's users, i.e., they shall not implement their own instances.

As previously mentioned, impure computations should be defined over a polymorphic composite interface rather than a concrete interface type for them to be seamlessly composed together using the `bind` combinator. As an example, FreeSpec provides `request`, a monadic helper which simply requests the handling of a primitive and returns back its result. `request` is defined as follows:

```
Class MayProvide (ix i : interface) :=
  { proj_p {t} (e : ix t) : option (i t)
  }.
Class Provide (ix i : interface)
   `{MayProvide ix i} :=
  { inj_p {t} (e : i t) : ix t
  ; proj_inf_equ {t} (e : i t)
    : proj_p (inj_p e) = Some e
  }.
Class Distinguish (ix i j : interface)
   `{Provide ix i} `{MayProvide ix j} :=
  { distinguish {t} (e : i t)
      : proj_p (i:=j) (inj_p e) = None
  }.
```

Figure 1. The `inj_p` and `proj_p` type classes hierarchy

```
Definition request `{Provide ix i} {t} (p : i t)
  : impure ix t :=
  request_then (inj_p p) local.
```

Albeit polymorphic, `ix` is constrained thanks to the argument written `` `{ Provide ix i } ``: this type class constraint ensures that `ix` contains at least the primitives of the interface `i`. We can rewrite the `get_inc` impure computation previously defined in a more generic way (using a do-notation *à la* Haskell to make the code more readable):

```
Definition get_inc `{Provide ix (STORE nat)}
  : impure ix nat :=
  do x <- request Get;
     request (Put (S x));
     pure x
  end.
```

While convenient, this type class-based mechanism prevents an impure computation to use two distinct implementations of the same interface. Indeed, the type class resolution mechanism is directed by the type constructor identifying the interface. The standard way to circumvent this limitation is to use newtypes to distinguish between the two implementations.

2.3 Interface Contracts

In addition to specifying impure computations, FreeSpec provides tools to verify them. To that end, we rely on a contract-based verification approach, wherein *interface contracts* allow for specifying (1) how to use an interface, and (2) what to expect from the results of its primitives. The definition of FreeSpec interface contract remains unchanged compared to previous work. For illustration purpose, we take the example of the `STORE` interface introduced earlier. On the one hand, a reasonable caller obligation can be to only retrieve the value of the global variable once it has been initialized. On the other hand, a common callee obligation for

```
Definition store_contract (s : Type)
  : contract (STORE s) (option s) :=
  {| witness_update :=
       fun ms _ e _ =>
         match e with
         Get => ms | Put x => Some x
         end
   ; caller_obligation :=
       fun ms _ e =>
         match e with
         Get => is_some ms | Put x => True
         end
   ; callee_obligation :=
       fun ms _ e x =>
         match e, x with
         | Get, x => match ms with
                     | Some s => s = x
                     | _ => True
                     end
         | Put _, _ => True
         end
  |}.
```

Figure 2. An example of interface contract for the STORE interface

such an interface would be that only the Put can modify the value of the global variable. As this example demonstrates, obligations are likely to evolve along the evaluation because of side-effects *e.g.*, an impure computation is not allowed to use Get until a first Put has been performed, and the expected result of Get changes after a call to Put.

As a consequence, interface contracts are defined against so-called *witness states*. A witness state gives an abstract account of the current state of the component, and is therefore updated after a primitive call. In FreeSpec, interface contracts are encoded with the contract type, defined as follows:

```
Record contract (i : interface) (Ω : Type) :=
  { witness_update (ω : Ω)
    : forall (t : Type), i t -> t -> Ω
  ; caller_obligation (ω : Ω)
    : forall (t : Type),  i t -> Prop
  ; callee_obligation (ω : Ω)
    : forall (t : Type), i t -> t -> Prop
  }.
```

where witness_update describes the effect of each primitive on the witness state ; caller_obligation describes which operations are allowed in a given witness state ; and finally callee_obligation characterizes the answers of each primitive under a given witness state.

In Figure 2, we propose a concrete definition for the contract previously discussed for the STORE interface. The witness state of this contract is of type option s, where the term None encodes the fact that Put has never been called, and Some x for a certain x of type s means the last call to Put has had x as a parameter.

Respectful Computations. The general intuition behind interface contracts is the following: as long as the callers satisfy their obligations, callees are expected to do as well.

From this perspective, we say that an impure computation is *respectful* of a contract when it always satisfies caller obligations under the hypothesis that callee obligations are met. Given a term x, the computation local x is always respectful of c in accordance to ω, since it does not use any primitives. Given now a primitive p and a continuation f, the impure computation request_then p f is respectful of c in accordance to ω if:

1. p satisfies c's caller obligations, and
2. given any x which is a correct result for p (w.r.t. c's callee obligation), f x is respectful of c in accordance to ω', where ω' is the witness after p's execution.

Before entering into the formal definitions of respectfulness, we need first to emphasize that, since the impure monad and polymorphic composite interfaces allow for specifying and composing impure computations together, an arbitrary impure computation may use other primitives which do not belong to the interface referred by c, or even use none of the primitives of this interface at all. We therefore use the MayProvide type class to generalize the three components of an interface contract. For instance, we can use the proj_p function to distinguish between relevant and irrelevant primitives, and update the witness state only in the former case:

```
Definition gen_witness_update `{MayProvide ix i}
    {Ω t} (c : contract i Ω) (ω : Ω)
    (p : ix t) (x : t) : Ω :=
  match proj_p i with
  | Some e => witness_update c ω e x
  | None => ω
  end.
```

Similarly, we generalize both caller_obligation and callee_obligation to handle an arbitrary composite interface ix, such that the predicates are automatically satisfied when considering a primitive of ix which cannot be projected to i. In other words, these proof obligations automatically exploit the fact that interfaces are *separated by construction*. Thus, they cannot interfere and we can separately reason about their correct usage even though a program can use several interfaces simultaneously.

Using these functions, we can reason about the respectfulness of an impure computation leveraging a polymorphic composite interface ix w.r.t. a contract about the interface i. In the previous iteration of FreeSpec [23], the predicate of respectfulness was not polymorphic. As a consequence, it was not possible to seamlessly reuse a proof of respectfulness for an impure computation using an interface i to prove

the respectfulness of a host computation which uses more interfaces. The respectfulness predicate is defined as follows:

```
Inductive respectful_impure `{MayProvide ix i} {t Ω}
    (c : contract i Ω) (ω : Ω)
  : impure ix t -> Prop :=
| respectful_local (x : t)
  : respectful_impure c ω (local x)
| respectful_request {u}
    (e : ix u) (f : u -> impure ix t)
    (* We must respect e's requirements on ω. *)
    (ocaller : gen_caller_obligation c ω e)
    (ocallee : forall (x : u),
     (* The answer validates the requirements. *)
     gen_callee_obligation c ω e x ->
     (* We continue with the next witness ω'. *)
     let ω' := gen_witness_update c ω e x in
     respectful_impure c ω' (f x))
  : respectful_impure c ω (request_then e f).
```

Asserting that an impure computation p is respectful of a contract c requires to prove that p will always satisfy c's caller obligation, under the sole hypothesis that c's callee obligation is satisfied. From this perspective, the `impure` monad is boilerplate, and FreeSpec provides a dedicated tactic called `prove_impure` to completely erase it. More precisely, `prove_impure` identifies the proof obligations related to `caller_obligation` necessary to conclude about a given impure computation respectfulness. To achieve that, `prove_impure` performs a delicate symbolic execution of the program, in the same spirit as the CFML condition generator [7]. The obligations that Coq **`auto`** command is not able to handle are then left for the user to solve.

As a consequence, reasoning about the respectfulness of an impure computation p eventually boils down to an enumeration of all the contexts under which p calls a primitive, and to a proof that each of these calls is licit. As p grows in complexity, this enumeration becomes cumbersome. Besides, p can be a composition (by means of the `bind` operator of the `impure` monad) of several smaller computations, and sometimes the same computation can be used several times. In such a case, in addition to a large number of proof obligations about `caller_obligation`, many of these obligations will be very similar. More precisely, to prove that `p >>= f` is respectful of a contract c for a witness state ω, we cannot reuse a proof that p is respectful of c for ω as-is, but rather need to enumerate the primitive calls of p once again.

To enable proof reuse, we need a way to reason about computation outcomes, both in terms of result and witness state updates. In FreeSpec, this is primarily achieved with the so-called *respectful runs*.

Respectful Runs. A respectful run (w.r.t. a contract c about an interface `i`) of an impure computation p encodes an execution of p during which both p and the callee which handles primitives of `i` satisfy their respective obligations. This predicate is supposed to occur as an hypothesis to reason about the behavior of an impure computation which is assumed to execute safely. From this hypothesis, we can typically decompose the impure computation as a sequence of requests and make explicit the intermediate witness states that occur after each effects. During this decomposition, the proofs of the conformance to the contract's callee and caller obligations are also made available.

Given a term `x`, and an initial witness ω, then there exists only one respectful run of the computation `local x`. The latter returns `x` as a result, and preserves the witness state ω since it does not perform any effectful operation.

Consider an initial witness ω, a primitive `p` which satisfies `c`'s caller obligations, and a continuation `f`. We say that "there exists a respectful run for `request_then p f` which leads to a result `y` and a witness state ω'" if (i) there is some result `x` for `p` which satisfies `c`'s callee obligations ; and (ii) there exists a respectful run for `f x` which leads to a result `y` and a witness state ω'. In other words, we make explicit the intermediate witness state ω'' which intervenes just before the request continuation is triggered.

```
Inductive respectful_run `{MayProvide ix i}
    {t Ω} (c : contract i Ω)
  : impure ix t -> Ω -> Ω -> t -> Prop :=
| run_local (x : t) (ω : Ω)
  : respectful_run c (local x) ω ω x
| run_request {u} (ω' : Ω)
    (p : ix u) (f : u -> impure ix t)
    (ω : Ω) (x : u) (y : t)
    (ocaller : gen_caller_obligation c ω p)
    (ocallee : gen_callee_obligation c ω p x)
    (rec :
     let ω'' := gen_witness_update c ω p x in
     respectful_run c (f x) ω'' ω' y)
  : respectful_run c (request_then p f) ω ω' y.
```

Probably because they are closely related, the difference between `respectful_impure` and `respectful_run` has proven to be difficult to grasp.

On the one hand, `respectful_impure c ω p` states that as long as the underlying callee satisfies its obligations, then p behaves in accordance with its own obligations as well. On the other hand, `respectful_run c p ω ω' x` states that there exists an execution of p (1) such that p and the underlying callee have both satisfied their obligations, (2) where the witness state was updated to ω', and (3) resulting in the production of the term `x`. It is possible to prove the existence of a respectful run for a computation p which is not always respectful of a contract c. Indeed, even if this computation does not behave in accordance with c's caller obligations in the general case, it may exist a callee whose results happen to make it behave respectfully occasionally. It is also possible to prove the existence of two respectful runs of p starting from the same witness state, but leading to the production of different results. We can prove, however, that if an impure computation p is respectful of a contract c, then

all its executions with a callee which satisfies its obligations w.r.t. c can be encoded as a respectful run.

Therefore, given a computation p and a continuation f, if we have (1) a proof that p is respectful of c for ω, and (2) a proof that, for every result x and witness state ω' that p execution can lead to, f x is respectful of c for ω', then we can conclude that p >>= f is respectful of c for ω *without being forced to enumerate the primitive calls made by* p.

```
Lemma respectful_bind_respectful_run
  `{MayProvide ix i} {t u Ω}
   (c : contract i Ω) (ω : Ω)
   (p : impure ix u) (f : u -> impure ix t)
   (trust : respectful_impure c ω p)
   (run : forall (x : u) (ω' : Ω),
       respectful_run c p ω ω' x
       -> respectful_impure c ω' (f x))
 : respectful_impure c ω (p >>= f).
```

The prove_impure tactic mentioned previously leverages this lemma when it encounters an opaque impure computation as the first parameter of a bind combinator. This results in the production of two goals. The first one —p is respectful— is left for the user to prove (typically by means of an already existing lemma). The second one —f x is respectful, under the hypothesis that x is the result of a respectful run of p— is recursively handled by the prove_impure tactic.

The respectful_run predicate is expected to be used exclusively as an hypothesis. A common pattern in FreeSpec developments is to provide general-purpose lemmas of the following form:

```
forall ω' x,
  respectful_run c p ω ω' x -> P ω' x
```

which can be later used in goals generated by prove_impure, in conjunction with lemmas of the following form:

```
forall ω x,
  P ω x -> respectful_impure c ω (f x)
```

To leverage hypotheses of the form respectful_run, we provide a dedicated tactic called unroll_respectful_run. The goal of this tactic is to destruct the run of p to expose to the user which primitives and associated results led to this run, as well as the obligations they satisfy.

Additionally, respectful_run can be used to exhibit interesting properties of respectful impure computations beyond their respectful usage of a given interface. For instance, coming back to the STORE interface and the store_contract contract, we can prove that a respectful computation always restores the value of the global mutable variable before returning its result with the following predicate:

```
Definition store_preserving {s a}
  `{MayProvide ix (STORE s)}
   (p : impure ix a) (x : a)
   (init final : option s)
 : respectful_run store_contract p init final x
   -> init = final.
```

Contracts Composition. Contracts can be conveniently composed thanks to the following operator.

```
Definition contract_plus
  `{Provide ix i, Provide ix j} {Ωi Ωj}
   (ci : contract i Ωi)
   (cj : contract j Ωj)
 : contract ix (Ωi * Ωj) :=
 {| witness_update :=
      fun ω _ e x =>
        (gen_witness_update ci (fst ω) e x,
         gen_witness_update cj (snd ω) e x)
  ; caller_obligation :=
      fun ω _ e =>
        gen_requirements ci (fst ω) e
        ∧ gen_requirements cj (snd ω) e
  ; callee_obligation :=
      fun ω _ e x =>
        gen_promises ci (fst ω) e x
        ∧ gen_promises cj (snd ω) e x
  |}.
```

There are two interesting properties of contract_plus that we want to emphasize. First, nothing prevents i and j to be the same interface. Second, contract_plus is commutative and associative. This entails that the way we compose a set of contracts together does not matter.

3 FreeSpec.Exec

3.1 Effectful semantics

The previous section has introduced impure computations which rely on an interface to implement the effectful primitives they use. For now, we have pushed back the problem of implementing impurity to another layer of the program. If we want to actually execute these impure programs, we cannot push back the implementation of impure primitives *ad infinitum*, we must complete them with an effectful semantics. We define this notion as follows.

```
CoInductive semantics (i : interface) : Type :=
| mk_semantics :
  (f : forall (t : Type), i t -> interp_out i t)
  : semantics i
with interp_out (i : interface) : Type -> Type :=
| mk_out {t} (x : t) (sem : semantics i)
  : interp_out i t.
```

A semantics for an interface i provides an interpretation for each primitive p of type i a by returning a value of type t as well as a new semantics. This new semantics corresponds to the previous semantics modified by the effects of p.

From the perspective of an interface i, a semantics is the stream of primitive results produced by an implementation of i. For this reason, it is defined as a coinductive object.

There are four ways to realize an effectful semantics: by simulation, by delegation, by extraction, and by interpretation. First, one can *simulate* the effectful semantics directly within Coq, e.g. a STORE can be implemented as follows:

```
CoFixpoint store {s} (init : s)
: semantics (STORE s) :=
  mk_semantics (fun (t : Type) (e : STORE s t) =>
    match e with
    | Get => mk_out init (store init)
    | Put next => mk_out tt (store next)
    end).
```

Second, the realization of a semantics for an interface `i` *can be delegated to another impure computation* which itself exploits another interface `j`. This technique allows for a hierarchical decomposition of large effectful systems as explained in our previous paper about FreeSpec. The present paper focuses on effectful programming-in-the-small rather than effectful programming-in-the-large [11]: for this reason, we omit the description of FreeSpec's notion of `component` which allows for modular development of certified large systems.

These first two realizations of semantics are useful to simulate or to organize effectful computations. However, when it comes to actually run the program, we need executable realizations of semantics.

3.2 Executable Realizations of Semantics

The extraction [24] is the standard way to turn a Coq development into a runnable executable. This method can be used with FreeSpec as well, except that the semantics of effects must be explicitly realized by OCaml expressions, e.g.:

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

CoFixpoint ocaml_semantics :=
  mk_semantics (fun {t} (x : CONSOLE t) =>
    match x with
    | Scan =>
      mk_out (ocaml_scan tt) ocaml_semantics
    | Echo s =>
      mk_out (ocaml_echo s) ocaml_semantics
    end).
```

This technique does not fit well the interactive workflow of Coq since it requires leaving the current user session to issue some batch commands to build an executable and to run it afterward. Each experimentation interrupts the interactive user session, which can be cumbersome in the long run.

FreeSpec.Exec is a Coq plugin that introduces a Vernacular command written `Exec t` that takes a term `t` of type `impure ix t` and executes it within Coq. This execution technique does not require the user to leave its interactive session and therefore smoothly integrates the user workflow. Here is how an Hello World program can be executed within Coq (`echo` being a wrapper for a `request` for the Echo primitive):

```
Definition hello `{Provide ix CONSOLE}
: impure ix unit :=
  echo "Hello, world!".
Exec hello.
```

The FreeSpec.Exec is meant to give a smooth development experience and to provide a simple interpretation mechanism, similar the Python interpreter for instance. Of course, for performance sensitive applications, we plan to provide a compilation scheme for FreeSpec based software.

3.3 Implementation

Inspired by Mtac [19, 36] and MetaCoq [2], FreeSpec.Exec is a bi-interpreter, i.e., an interpreter that interleaves two interpreters. One of these two interpreters is the reduction engine of Coq: when it is provided a term of type `impure ix t`, this interpreter makes the head constructor appear using the weak-head reduction strategy. Then, it is the turn of the other interpreter that we wrote to deal with the impure parts of the computation: it has an evaluation rule for each constructor of `impure`. When the constructor is `Pure`, the computation is done. Otherwise, when the constructor is `Request`, this second interpreter performs the requested operation using the impure facilities of OCaml and calls the first interpreter back by passing the continuation instantiated with the operation answer.

Definition 3.1 (Evaluation strategy of FreeSpec.Exec)**.** Let `t` be of type `impure ix t`.

1. Compute `w`, the weak-head normal form of `t`.
2. If `w` is `Pure u`, returns `u`.
3. If `w` is `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)`.

3.4 FreeSpec.Exec plugin system

FreeSpec.Exec is a Coq plugin but it also enjoys a plugin system of its own. Thanks to this plugin system, any Coq library that defines a FreeSpec interface can also provide some OCaml code to concretely realize the effects of this interface. This piece of code is registered quite easily by calling the `register_interface` function as shown by the implementation of the `Console` interface as found in FreeSpec standard library:

```
let path = "freespec.stdlib.console"

let install_interface =
  let scan = function
    | [] -> string_to_coqstr (read_line ())
    | _ -> assert false
  and echo = function
    | [str] -> print_bytes (bytes_of_coqstr str);
               coqtt
    | _ -> assert false
  in
  register_interface path [
    ("Scan", scan); ("Echo", echo)
  ]
```

Exec provides a basic set of conversion functions to turn OCaml values into Coq terms, and conversely. Hence, the programmer can easily wrap OCaml functions to lift them to effect handlers semantics in Coq. For FreeSpec, Exec therefore plays the role of a Foreign Function Interface to OCaml. Since Coq 8.10, the use of the `Register` vernacular command is required in order for plugins to interact with named constants. For instance, for the `CONSOLE` interface, the following statements are necessary:

```
Register CONSOLE as freespec.stdlib.console.type.
Register Scan as freespec.stdlib.console.Scan.
Register Echo as freespec.stdlib.console.Echo.
```

4 MiniHTTPServer

`MiniHTTPServer` is a "minimal web server" implemented and verified in Coq thanks to FreeSpec. `MiniHTTPServer` is inspired from the `SimpleHTTPServer` module of Python: its purpose is to serve static files over HTTP. In its current state, `MiniHTTPServer` works as follows. First, it opens a TCP socket, and waits for an incoming connection. Once established, `MiniHTTPServer` waits for a TCP packet, then it parses the content of this packet as a HTTP request to extract a resource identifier (*e.g.* `index.html`, `css/style.css`, etc.). If the HTTP request is well-formed, and if the requested resource exists in the file system, `MiniHTTPServer` fetches its content, and sends it back to the client as a valid HTTP response. Otherwise, it sends back the relevant HTTP status (that is, 400 if the request is ill-formed or 404 is the resource does not exist). `MiniHTTPServer` canonicalizes resource identifiers and prefixes it with a base directory (*e.g.*, `../../etc/passwd` becomes `/srv/etc/passwd`) to protect itself against directory traversal attacks [15]. In its current state, `MiniHTTPServer` will only accept a finite number of connections before exiting gracefully, due to the inductive nature of the `impure` type.

There is plenty of room for improvement, but our goal with `MiniHTTPServer` is neither features completeness, nor efficiency. On the contrary, we aim to keep this project relatively small and simple to be easily understood, while making it an appealing demonstrator for the FreeSpec framework.

4.1 Implementing a web server in Coq

The `MiniHTTPServer` implementation can be divided into three elementary tasks: (1) waiting for arbitrary incoming inputs and sending back outputs to TCP sockets (thanks to the `TCP` interface), (2) interpreting incoming packets as HTTP requests, and (3) fetching resources content from the file system (thanks to `FILESYSTEM` interface).

Handling TCP Connections. The `tcp_server` computation is the only impure computations of `MiniHTTPServer` which directly uses the following `TCP` interface:

```
Inductive TCP : interface :=
| NewTCPSocket
  : string -> TCP socket_descriptor
| ListenIncomingConnection
  : socket_descriptor -> TCP unit
| AcceptConnection
  : socket_descriptor -> TCP socket_descriptor
| ReadSocket
  : socket_descriptor -> TCP string
| WriteSocket
  : socket_descriptor -> string -> TCP unit
| CloseTCPSocket
  : socket_descriptor -> TCP unit.
```

More precisely the TCP-related impure computations called by `tcp_server` are all simple wrappers upon a call of `request` with a `TCP` primitive. `tcp_server` takes two arguments: the number of incoming connections it has to handle, and an arbitrary handler which computes responses to send back to the clients. This function is used to define the HTTP server as follows:

```
Definition http_server
  `{Provide ix FILESYSTEM, Provide ix TCP}
  : impure ix unit :=
  tcp_server (tcp_handler [Dirname "tmp"]).
```

where `tcp_handler` is interpreting TCP packets as HTTP requests under the `"tmp"` directory. Readers familiar with Haskell or other purely functional languages relying on monads to encode side-effects will find the implementation of `tcp_server` (in Figure 3) familiar.

```
Definition tcp_server `{Provide ix TCP}
    (handler : string -> impure ix response) (n : nat)
  : impure ix unit :=
  do server <- new_tcp_socket "localhost:8080";
     listen_incoming_connection server;
     repeatM n do
       client <- accept_connection server;
       req <- read_socket client;
       res <- handler req;
       write_socket client res;
       close_socket client;
     end
     close_socket server
  end.
```

Figure 3. The `tcp_server` definition

Interpreting HTTP Requests. Interpreting incoming HTTP requests is two-fold.

First, `MiniHTTPServer` parses the raw packets, in order to extract resource identifiers from a `GET` HTTP request. The parsing function is implemented with a homegrown parser combinators library we called `coq-comparse` similar to Haskell `parsec` package or Rust `nom` crate[3]. Then,

[3] `coq-comparse` source code can be downloaded at https://github.com/ANSSI-FR/coq-comparse. It is distributed under the terms of the GPLv3 license.

MiniHTTPServer canonicalizes this resource identifier to remove the . and .. special directories. This second step is mandatory to avoid MiniHTTPServer to be subject to traversal directories attacks.

Serving Static Files. Once the resource identifier has been extracted and sanitized, MiniHTTPServer checks if the resource exists. In such a case, it fetches its content from the file system, then crafts a valid HTTP response for the client. Otherwise, it sends back to the client the famous "404 Not found" HTTP error.

To achieve this, MiniHTTPServer relies on the FILESYSTEM interface, defined as follows:

```
Inductive FILESYSTEM : interface :=
| Open (path : string) : FILESYSTEM file_descriptor
| FileExists (file : string) : FILESYSTEM bool
| Read (file : file_descriptor) : FILESYSTEM string
| Close (file : file_descriptor) : FILESYSTEM unit.
```

4.2 Verifying MiniHTTPServer implementation

As discussed in Section 2, FreeSpec introduces two key components to reason about impure computations: (1) interface contracts which assign proof obligations to callers and callees, and (2) tactics to erase the impure monad. We are convinced, after using FreeSpec for verifying increasingly complex use cases, that (2) is mandatory for (1) to be tractable. This technique based on a verification condition generator written in Ltac has already proven its relevancy in other tools for program verification in Coq, *e.g.*, CFML [7].

We now explain how we have used FreeSpec in order to prove two properties of MiniHTTPServer, that is our server correctly manages its file descriptors (*e.g.*, it does not try to read a file with a closed file descriptor) and it does not leave any file descriptors open. Albeit simple, these two properties are good candidates to demonstrate FreeSpec formalism and automation capabilities benefits.

Interface Contract. As a mandatory first step, we define an interface contract to encode the two targeted properties of MiniHTTPServer. As a reminder, an interface contract for an interface i is parameterized with witness states and consists in three components: two predicates to specify the callers and callees obligations, and a function to update the witness state after each primitive call.

We define the witness state of our contract as the set of currently open file descriptors. The witness state is only used for reasoning about an impure computation, and is not involved during the execution of an impure computation. It is therefore similar to a phantom parameter as found for instance in Dafny [22] or Why3 [13]. Since the choice of the witness state representation is not subject to any performance or memory consumption considerations, we use functions of type t -> bool to encode sets of terms of type t. In our experience, this representation is straightforward and eases the reasoning on set manipulation, partly because Coq efficiently reduces them in practice.

The definitions of three components of our interface contract fd_set_contract are given in Figure 4:

- When the witness state update function is given a primitive of the form Open path resulting in a file descriptor fd (resp. a primitive of the form Close fd), it inserts (resp. removes) fd from the witness state. The other primitives leave the witness state untouched.
- Caller obligations constrain file descriptors passed as arguments to primitives to inhabit the witness state.
- Callee obligations only constrain the Open primitive to produce file descriptors that do not inhabit the witness state. Notice that this obligation does not forbid callees to reallocate closed file descriptors. If we wanted to prevent such a behavior, we could have enforced a stricter freshness policy by using a dedicated type in place of bool with three values Fresh, Open and Closed, and modify the callee obligations accordingly.

Theorems Statements. On the one hand, we use the predicate respectful_impure to model the property that our server correctly manages its file descriptors.

```
Theorem fd_set_contract_http_server
  `{Provide ix FILESYSTEM, Provide ix TCP}
   (s : fd_set)
: respectful_impure fd_set_contract s http_server.
```

On the other hand, we use the respectful_run predicate to model the property that MiniHTTPServer closes every file it opens. To that end, we first define a predicate we call fd_set_preserving, following the same approach as the store_preserving predicate we have discussed earlier (in Subsection 2.3).

```
Definition fd_set_preserving {t}
  `{MayProvide ix FILESYSTEM}
   (p : impure ix t) :=
 forall (s s' : fd_set) (x : t),
   respectful_run fd_set_specs p s s' x
   -> forall fd, s fd = s' fd.
```

The fd_set_preserving predicate states that, for every respectful run of p, any file descriptor open (resp. closed) before the execution of p remains open (resp. closed) afterwards. As a consequence, if p opens a file, it closes it before the end of its execution.

Therefore, we can use the fd_set_preserving predicate to encode the fact that http_server closes any file it opens.

```
Theorem fd_set_preserving_http_server
  `{Provide ix FILESYSTEM, Provide ix TCP}
: fd_set_preserving http_server.
```

Conducting proofs within FreeSpec. We now give a high-level overview of how proofs related to impure computations are conducted using FreeSpec. The fd_set_contract contract only defines obligations for the FILESYSTEM interface.

```
Definition fd_set : Type := file_descriptor -> bool.

Definition fd_set_update (S : fd_set) (a : Type) (e : FILESYSTEM a) (x : a) : fd_set :=
  match e, x with
  | Open _, fd => add_fd S fd
  | Close fd, _ => del_fd S fd
  | Read _, _ => S
  | FileExists _, _ => S
  end.

Inductive fd_set_caller_obligation (S : fd_set) : forall (a : Type), FILESYSTEM a -> Prop :=
| fd_set_open_caller (p : string)
  : fd_set_caller_obligation S file_descriptor (Open p)
| fd_set_read_caller (fd : file_descriptor) (is_member : member S fd)
  : fd_set_caller_obligation S string (Read fd)
| fd_set_close_caller (fd : file_descriptor) (is_member : member S fd)
  : fd_set_caller_obligation S unit (Close fd)
| fd_set_is_file_caller (p : string)
  : fd_set_caller_obligation S bool (FileExists p).

Inductive fd_set_callee_obligation (S : fd_set) : forall (a : Type), FILESYSTEM a -> a -> Prop :=
| fd_set_open_callee (p : string) (fd : file_descriptor) (is_absent : absent S fd)
  : fd_set_callee_obligation S file_descriptor (Open p) fd
| fd_set_read_callee (fd : file_descriptor) (s : string)
  : fd_set_callee_obligation S string (Read fd) s
| fd_set_close_callee (fd : file_descriptor) (t : unit)
  : fd_set_callee_obligation S unit (Close fd) t
| fd_set_is_file_callee (p : string) (b : bool)
  : fd_set_callee_obligation S bool (FileExists p) b.

Definition fd_set_contract : contract FILESYSTEM fd_set :=
  {| witness_update := fd_set_update
   ; caller_obligation := fd_set_caller_obligation
   ; callee_obligation := fd_set_callee_obligation
  |}.
```

Figure 4. The interface contract `fd_set_contract`.

The unique place where `MiniHTTPServer` uses this interface is after it has extracted a canonical resource identifier from an incoming well-formed HTTP request. In such a case, `MiniHTTPServer` first checks if the resource identifier is a path to an existing file with a primitive called `FileExists`. When this primitive produces the value `true`, the server opens, reads the content, then closes the file.

From our perspective, this routine of `MiniHTTPServer` is critical since this is the place where something bad may happen: if a bug prevents serving the requested file content, the web server is useless. The verification effort amounts to prove a statement of the form:

```
forall (s : fd_set),
  respectful_impure
    fd_set_contract s
    (do var isf <- file_exists path in
        if (isf : bool)
        then do
          var fd <- open path in
          var content <- read fd in
          close fd;
          pure (Some content)
        end
        else pure None
    end)).
```

In this code snippet, the treatment of the file content is enclosed by the opening and the closing of the file. In such a scenario, FreeSpec users are entitled to expect an automatic proof using the tactic `prove_impure` mentioned earlier. Interestingly enough, FreeSpec is able to solve this goal automatically when the appropriate `Hint`s has been registered. In a nutshell, the tactic proceeds as follows:

1. `prove_impure` uses `cbn` to reduce the current goal, and inspects the current form of the impure computation only to realize the latter uses the `FileExists path` primitive; `prove_impure` generates a subgoal to prove

this primitive satisfies `fd_set_contract`'s caller obligation and if **`auto`** does not succeed in solving it (it does with the appropriate **`Hint`**), `prove_impure` leaves this goal to be solved by the user.
2. It applies the constructor of `respectful_impure` to the main goal, and calls itself recursively.
3. The current form of the impure computation to prove is now **`if`** `isf` **`then`** `...` **`else`** `...`; `prove_impure` uses **`destruct`** to explore both alternatives. It calls itself recursively.
4. Assuming `isf = true`:
   a. the impure computation now uses the primitive **`Open`**. `prove_impure` produces a dedicated subgoal, applies the appropriate constructor to the main goal, and recursively calls itself.
   b. In the main goal, the fact that the result of **`Open`** satisfies `fd_set_contract` contract now appears as an hypothesis. We therefore know that (1) the file descriptor was not in the witness state before, and also (2) the file descriptor is now in the new witness state thanks to the witness state update function. Besides, the impure computation now uses the primitive `Read`, so `prove_impure` yet again generates a subgoal related to the caller obligation of `Read`. This subgoal is easily solved with the hypothesis (2).
   c. Then comes the primitive **`Close`**, which is handled exactly like `Read` since they have the same caller obligation.
5. Assuming now `isf = false`, the impure computation ends without further usage of the `FILESYSTEM` primitives.

Since we have now formally established the respectfulness of `MiniHTTPServer` simple "open, read, then close" routine, we can reason about its outcomes. More precisely, we show this routine satisfies the `fd_set_preserving` predicate.

```
forall (s s' : fd_set) (res : unit),
  respectful_run fd_set_preserving
    (do var isf <- file_exists path in
        if (isf : bool)
        then do
          var fd <- open path in
          var content <- read fd in
          close fd;
          pure (Some content)
        end
        else pure None
    end) s s' res -> forall fd, s fd = s' fd.
```

As mentioned earlier, the `respectful_run` is here used as an hypothesis. We use the tactic `unroll_respectful_run` to explore the set of respectful runs automatically. In this context, we consider two scenarios:

1. There is one respectful run where `FileExists` returns `false`, and the computations ended. In such a case, we can conclude `s fd = s' fd` for all `fd`, since `s = s'` by definition of the witness update function.
2. The rest of the respectful runs of our routine proceeds as follows: the `FileExists` primitive has returned `true`. This leads to the primitive **`Open`** being called, and its result being passed as an argument for the primitives `Read` and **`Close`**.
   a. In practice, after the call to **`Open`**, the returned file descriptor is added to the witness state.
   b. Then, the call to `Read` leaves the witness state
   c. Finally, after the call to **`Close`**, the file descriptor previously returned by **`Open`** is removed from the witness state. We can conclude about `s fd = s' fd` for any `fd` because, thanks to `fd_set_contract` callee obligations, we know the file descriptor returned by **`Open`** was not in the witness state to begin with, so adding it and removing it eventually leaves the witness state in the same state it was initially.

The rest of `MiniHTTPServer` only uses the file system interface *via* this routine. To reuse our two previous proofs, we thereafter treat this computation as an opaque constant, meaning `prove_impure` and `unroll_respectful_run` will not unfold them (*i.e.*, the tactic `prove_impure` will leverage the `respectful_bind_respectful_run` lemma). Besides, `prove_impure` proves the subgoals related to primitives irrelevant to the targeted contract automatically (they are always satisfied, by definition of `gen_caller_obligation` and `Distinguish`).

That being said, FreeSpec tactics cannot automatically solve the two main theorems we aim to prove. These tactics erase the FreeSpec internal boilerplate to *sequentialize* calls of primitives or opaque impure computations, but they are not equipped to deal with conditional branching in the general case. In Coq, this is modeled with datatypes inductive principles (and the **`match`** `...` **`with`** `...` **`end`** syntactic sugar). With the notable exception of the `bool` type, FreeSpec tactics will leave to the user the choice of a strategy to reason about inductive principles (*e.g.*, performing an induction, a case analysis, etc.).

For `MiniHTTPServer`, there are two uses of an induction principle: (1) with `sum` type to reason about what to do with the result of the HTTP request parsing which can be either a parsing error or a resource identifier, and (2) with the `nat` type whether the server can still accept incoming connections or not (inside the `repeatM` helper appearing in Figure 3). We can tackle case (1) with a simple case analysis. Case (2) requires more work, but interestingly we can provide general-purpose lemmas to be reused not only for our server, but for any impure computation which relies on `repeatM`.

4.3 Executing `MiniHTTPServer`

As we discussed in Section 3, FreeSpec introduces a Coq plugin called FreeSpec.Exec, which acts as a callee for an extensible set of interfaces. Therefore, it is capable of executing impure computations which uses these interfaces directly within Coq. This allows us to run our server in a `coqtop` session and to interact with it with a standard web browser.

In theory, this execution model should not be significantly less efficient than other interpreted languages, *e.g.* Python. However, the current implementation of `MiniHTTPServer` suffers from the impractical representation of Coq strings. Indeed, the standard strings of Coq are implemented as list of ASCII characters which are themselves made of 8 booleans. This limitation will hopefully be addressed by the forthcoming integration of native arrays in Coq's internal term representations.

4.4 Discussion

In this section, we have implemented, verified w.r.t. an interface contract about the `FILESYSTEM` contract, and executed a minimal HTTP server. This development is made of 600 lines of specifications and 350 lines of proof scripts.

Albeit `MiniHTTPServer` remains a proof of concept in its current state, and is in no mean usable in a production environment, the project has been highly instructive.

Coq as a Programming Language. Programming languages are tools software developers use to build software components. In its core and to this day, Coq remains a theorem prover rather than a development environment. The fact that there is no canonical way to write a "Hello, world!" program using Coq by the end of 2019 (more than 30 years after its initial release) is a clear evidence of this fact. As a consequence, developers seeking to use Coq as their primary tool to write software expose themselves to the frustration of having to implement a lot of utilitary functions other languages tend to integrate in their standard library. Besides, they will probably feel in need for a better tooling ecosystem (*e.g.*, to this day, `coqdoc` is more a source pretty printer than a documentation tool comparable to `haddock` for Haskell or `cargo doc` for Rust). We believe these issues are not inherent to Coq, but rather the consequences of the Coq community priorities. Interestingly, turning Coq into a "real" programming language has been an increasing trend in recent years, and as a consequence Coq 8.10 has introduced native integers (and native array support is planned).

As a development environment, Coq definitely has unique and valuable features to put forth beyond allowing to assert the correctness of the produced software. The interactive development style allowed by `coqide` and its siblings combined with the Proof mode —where users can easily interact with and manipulate Coq terms— provide a very pleasant programming experience. It is reminiscent of the Common Lisp way of programming, based on a powerful REPL and hot code reloading, but is more robust in practice. Besides, Claret *et al.* has demonstrated how the Proof mode can be used to turn Coq into a powerful debugger for impure computations [10].

Certain properties of the Calculus of Constructions, which may seem limiting at first, can easily be advertised as selling points. For instance, the fact that Coq requires functions to terminate should be seen as a desirable feature, rather than an inconvenient necessity to avoid logic inconsistency. From `MiniHTTPServer` perspective, for instance, we know by construction that the handler we have defined to handle incoming TCP packets will terminate (provided that the unverified parts implemented in OCaml terminate, which is reasonable since the OCaml interpreter is defined by induction and since the OCaml primitives are simple system calls). Of course, the same requirement has forced us to limit the incoming connections our server accepts to a finite number, while a daemon by definition is expected to run indefinitely. We believe non-termination as the exception is a desirable property (similarly to Rust favoring immutability by default). We also ambition to provide abstractions to specify, verify and execute infinite computations in FreeSpec, but the best approach to achieve this goal is still unclear to us. The fact that the semantics of an interface is coinductive paves the way for obtaining non-termination thanks to a global loop, in the spirit of the so-called Folk theorem [16]. Anyway, each evaluation of an impure computation will still continue to terminate.

FreeSpec as a Verification Framework. Beyond the satisfaction to implement a web server run by `coqc`, the main motivation for `MiniHTTPServer` was to provide a large use case for FreeSpec updated formalism and verification framework. In practice, the properties we have aimed to prove in Subsection 4.2 are straightforward. In our opinion, the interface contract we have defined to encode these properties is easy to read, and reading the code of our web server is sufficient to be convinced `MiniHTTPServer` is indeed respectful. What we find interesting —and encourage us to consider this experience to be a success— is that the size of the proofs necessary to assert the respectfulness of our web server reflects that fact.

Besides, because interface contracts are defined alongside interfaces, rather than embedded into them, we can in practice introduce new contracts, then prove the respectfulness of `MiniHTTPServer` w.r.t. these contracts *without the need to modify existing proofs*. Thanks to the contract composition operator (see Subsection 2.3), we can also reuse our previous proofs to prove complementary result. The validity of the overall approach has been discussed in the original paper presenting FreeSpec [23].

FreeSpec.Exec and Dependent Types. Coq dependent types allow for encoding the specification of a function directly into the types of its arguments and its expected result. In

Coq, such a function will necessarily have to compute not only a resulting term, but also a proof of correctness for this term. Several approaches are possible to write such algorithms, most notably the Russel framework developed by Sozeau [31].

The computation of proofs in addition to regular results tends to dramatically reduce the performance of Coq evaluation mechanisms, most commonly because theorems and lemmas are traditionally defined as opaque constants, which leads to the production of huge terms even for simple computations. To our opinion, it remains too easy to write functions which cannot be evaluated in practice, and too hard to diagnose the origin of the issue when such scenario arises.

Avoiding dependent types for pure functions used by an impure computations (*i.e.*, favoring "regular" functions with separate lemmas) may appear as a reasonable compromise, but this is not always possible in practice, as we dolorously discovered during the development of the project `MiniHTTPServer`. Our parsing library features two recursive functions which cannot be defined by means of structural recursions. In such a case, the approach imposed by Coq is to rely on "well-founded recursion", which imposes the computation of a proof at each recursion step. Our experiments has tended to show that the weak-head normal form reduction strategy used by FreeSpec.Exec to find the next primitives to execute behaves poorly with such functions. On the contrary, the normal form reduction strategy later used to reduce the primitive terms provides way better performances. We have been able to improve `MiniHTTPServer` performance by leveraging the internals of FreeSpec.Exec to our advantage, by introducing an identity primitive (*i.e.*, of type `forall {a : Type}, a -> IDENTITY a`). This primitive can be used to force Coq to evaluate a given term using the normal form reduction strategy (in the context of `MiniHTTPServer`, the parsing of incoming HTTP requests).

Readers familiar with Haskell may find this situation reminiscent of the process of optimizing Haskell programs by locally enabling strictness (using the `!` annotation).

5 Related Work

FreeSpec tackles two questions already studied in the literature: how to reason about effectful computations in type theory? how to organize the verification of large systems?

5.1 Effecful computations in type theory

The DeepSpec project recently introduced a new representation of effecful programs in Coq with the so-called *interaction trees* [21, 35]. This approach shares a lot of similarities with ours as both are variants of the Free monad. However, interaction trees represent effectful computations through a *coinductive* datatype while, in contrast, `impure` computations are defined inductively. Hence, interaction trees denotes potentially diverging computations while `impure` computations always converge. Interaction trees are therefore expressive enough to express general recursion and mutual recursion between effectful computations. This expressiveness comes at a cost: interaction trees do not compute inside Coq which makes reasoning about them less convenient. By contrast, `impure` computations can compute inside Coq since their realization is a mere inductive function. This implies that FreeSpec is not limited to equational reasoning as Interaction Trees are: in FreeSpec, a proof can be conducted by symbolic execution of the impure program (as demonstrated by the macro `prove_impure`). This usually results in shorter proofs.

In addition, we have explained in Section 4.4 that FreeSpec can in theory represent diverging computations too thanks to coinductive semantics but we are still looking for the right design to integrate primitives for non-termination and the corresponding reasoning principles.

Using interaction trees, the DeepSpec project also certified a web server [21]. For the moment, `MiniHTTPServer` cannot be seriously challenge this achievement for many reasons. First, the DeepSpec web server is a realistic C program with necessarily better performances than `MiniHTTPServer`'s. Second, the verification challenge of this project is more difficult since it tackles the problem of interleaved interactions between the clients and the server through multiple connections. Third, it is based on an interface to a CertiKOS' socket model which is quite sophisticated.

Dylus *et al.* [9, 12] investigate the problem of reasoning about effectful computations in Coq where effects are modeled using monads. Like us, their final representation for effectful computations is based on a free monad but the parameterization of this free monad is slightly different. Instead of parameterizing the free monad with a parameterized type as in FreeSpec, they use a container [34] whose role is to abstract away a functor. It is unclear if this approach can model as many effects since there is no notion of effecful semantics. Besides, this piece of work does not tackle the problem of contract verification and composition.

Maillard *et al.* [26] recently showed that monad morphisms can be used to build very general relation between computational monads and specification monads. In FreeSpec, the relationship between an impure computation and its specification is implemented at the meta-level by the `prove_impure` tactic, which indeed follows the structure of the (free) monadic computations just like a monad morphisms would do, except that FreeSpec provides no guarantee that the tactic will actually succeed at removing the monadic layer of the computations. Therefore, we would probably gain more generality and more confidence into the program logic of FreeSpec by transporting these ideas in our framework. Besides, we would then be able to offer other standard verification techniques, typically based on predicate transformers [33].

Jomaa *et al.* have developed a protokernel (*i.e.*, a minimal microkernel) in Coq, by means of a state monad equipped

with pre- and post-conditions to enable Hoare Logic reasoning. On the contrary, FreeSpec contracts are not embedded inside impure computations, but defined as separate objects against interfaces. This means FreeSpec users can easily modify or add contracts, without breaking existing development. To execute their microkernel, they rely on an ad-hoc translation process to turn their monadic specification into a readable C program.

Algebraic effects and effect handlers led to a lot of research about verification of programs with side effects [5, 6], but to our surprise, we did not find any approach to write and verify programs with effects and effect handlers written for Gallina. However, other approaches exist. Ynot [28] is a framework for the Coq proof assistant to write, reason with and extract Gallina programs with side effects. Ynot side effects are specified in terms of Hoare preconditions and post-conditions parameterized by the program heap, and does not dissociate the definition of an effect and properties over its realization. To that extent, FreeSpec abstract specification is more expressive (thanks to the abstract state) and flexible (we can define more than one abstract specification for a given interface). Claret *et al.* have proposed Coq.io, a framework to specify and verify interactive programs in terms of use cases [10]. The proofs rely on *scenarios* which determine how an environment would react to the program requests. These scenarios are less generic and expressive than FreeSpec abstract specifications, but they are declarative and match a widely adopted software development process. They may be easier to read and understand for software developers.

Previous approaches from the Haskell community to model programs with effects using Free monads [3, 17, 20] are the main source of inspiration for FreeSpec.

5.2 Verification of large systems

FreeSpec's concept of abstract specification takes its root into the seminal work of Parnas [29] in which the author states that "the main goal is to provide specifications sufficiently precise and complete that other pieces of software can be written to interact with the piece specified without additional information. The secondary goal is to include in the specification no more information than necessary to meet the first goal". While Parnas [29] considers a module as "a device with a set of switch inputs and readout indicators", we extend the interaction with a module to be based on any type, including higher-order types, like functions. Moreover, FreeSpec's impure computations are first-class while modules Parnas [29] are second-class: thus, more composition patterns are possible within FreeSpec. Parnas [29] "do not insist that machine testing be done, only that it could conceivably be done": FreeSpec (and other similar systems) shows that machine testing of interface specification can now be done, thanks to the advent of proof assistants.

Contract-based software development[27] or verification[4, 14, 25] introduced many reasoning mechanisms to verify object-oriented systems, especially to enforce as much proof reuse as possible along class hierarchies. Dealing with object-oriented programming mechanisms lead to the introduction of concepts – like contravariant subtyping relations or ownerships – which are hard to grasp for software engineers, the aim of FreeSpec is to avoid as much of this complexity by avoiding late binding, indirect recursions or implicitly shared states. On the contrary, the "default" computational mechanisms offered by Coq and FreeSpec are simpler to verify than the ones of imperative settings, typically like the object-oriented systems. From that perspective, FreeSpec resembles the B-method [1] and it is also designed to enforce a refinement-based method of verification. Finally, FreeSpec also share this idea of restricting computational mechanisms with FoCaLiZe [30], a proof environment where proofs are attached to components and where programs are "functional programs with some object-oriented features".

Kami [8] shares many concepts with FreeSpec, but implements them in a totally different manner: impure computations are defined as labeled transition systems and can be extracted into FPGA bitstreams. Kami is hardware-specific, thus is not suitable to reason about systems which also include software components.

6 Future Work & Conclusion

In this article, we have presented the latest iteration of FreeSpec, our framework for implementing, specifying, verifying and as of now executing impure computations in Coq. The changes we have operated in our formalism enable a better composition of proofs about impure computations, which we believe is a key feature to consider writing certified programs and libraries with FreeSpec. To challenge our framework, we have also developed `MiniHTTPServer`, a minimal HTTP server written almost exclusively in Coq.

In a near future, we aim to provide an "IO standard library" (*e.g.*, a POSIX client library) including a collection of general-purpose interface contracts, so that FreeSpec users can build certified applications and libraries within Coq. We also plan to provide an efficient execution model for FreeSpec.

References

[1] Jean-Raymond Abrial and Jean-Raymond Abrial. 2005. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press.

[2] Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. 2018. Towards Certified Meta-Programming with Typed Template-Coq. In *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings (Lecture Notes in Computer Science)*, Jeremy Avigad and Assia Mahboubi (Eds.), Vol. 10895. Springer, 20–39. https://doi.org/10.1007/978-3-319-94821-8_2

[3] Heinrich Apfelmus. 2010. The `operational` package. https://hackage.haskell.org/package/operational.

[4] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. 2011. Specification and verification: the Spec# experience. *Commun. ACM* 54, 6 (2011), 81–91. https://doi.org/10.1145/1953122.1953145

[5] Andrej Bauer and Matija Pretnar. 2015. Programming with Algebraic Effects and Handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015), 108–123.

[6] Edwin Brady. 2014. Resource-dependent algebraic effects. In *International Symposium on Trends in Functional Programming*. Springer, 18–33.

[7] Arthur Charguéraud. 2011. Characteristic formulae for the verification of imperative programs. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 418–430. https://doi.org/10.1145/2034773.2034828

[8] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, et al. 2017. Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 24.

[9] Jan Christiansen, Sandra Dylus, and Niels Bunkenburg. 2019. Verifying effectful Haskell programs in Coq. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019*, Richard A. Eisenberg (Ed.). ACM, 125–138. https://doi.org/10.1145/3331545.3342592

[10] Guillaume Claret and Yann Régis-Gianas. 2015. Mechanical Verification of Interactive Programs Specified by Use Cases. In *Proceedings of the Third FME Workshop on Formal Methods in Software Engineering*. IEEE Press, 61–67.

[11] Frank DeRemer and Hans H. Kron. 1976. Programming-in-the-Large Versus Programming-in-the-Small. *IEEE Trans. Software Eng.* 2, 2 (1976), 80–86. https://doi.org/10.1109/TSE.1976.233534

[12] Sandra Dylus, Jan Christiansen, and Finn Teegen. 2019. One Monad to Prove Them All. *Programming Journal* 3, 3 (2019), 8. https://doi.org/10.22152/programming-journal.org/2019/3/8

[13] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 - Where Programs Meet Provers. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science)*, Matthias Felleisen and Philippa Gardner (Eds.), Vol. 7792. Springer, 125–128. https://doi.org/10.1007/978-3-642-37036-6_8

[14] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2013. PLDI 2002: Extended static checking for Java. *SIGPLAN Notices* 48, 4S (2013), 22–33. https://doi.org/10.1145/2502508.2502520

[15] Víctor H. García, Raúl Monroy, and Maricela Quintana. 2006. Web Attack Detection Using ID3. In *Toward Category-Level Object Recognition (Lecture Notes in Computer Science)*, Jean Ponce, Martial Hebert, Cordelia Schmid, and Andrew Zisserman (Eds.), Vol. 4170. Springer, 323–332. https://doi.org/10.1007/978-0-387-34749-3_34

[16] David Harel. 1980. On Folk Theorems. *Commun. ACM* 23, 7 (1980), 379–389. https://doi.org/10.1145/358886.358892

[17] Ralf Hinze and Janis Voigtländer (Eds.). 2015. *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*. Lecture Notes in Computer Science, Vol. 9129. Springer. https://doi.org/10.1007/978-3-319-19797-5

[18] Inria. [n.d.]. The Coq Proof Assistant. https://coq.inria.fr/.

[19] Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and Derek Dreyer. 2018. Mtac2: typed tactics for backward reasoning in Coq. *PACMPL* 2, ICFP (2018), 78:1–78:31. https://doi.org/10.1145/3236773

[20] Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 94–105.

[21] Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to interaction trees: specifying, verifying, and testing a networked server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, Assia Mahboubi and Magnus O. Myreen (Eds.). ACM, 234–248. https://doi.org/10.1145/3293880.3294106

[22] K. Rustan M. Leino. 2017. Accessible Software Verification with Dafny. *IEEE Software* 34, 6 (2017), 94–97. https://doi.org/10.1109/MS.2017.4121212

[23] Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. 2018. Modular Verification of Programs with Effects and Effects Handlers in Coq. In *22st International Symposium on Formal Methods (FM 2018)*. Springer.

[24] Pierre Letouzey. 2008. Extraction in Coq: An Overview. In *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings (Lecture Notes in Computer Science)*, Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe (Eds.), Vol. 5028. Springer, 359–369. https://doi.org/10.1007/978-3-540-69407-6_39

[25] Barbara Liskov and Jeannette M. Wing. 1994. A Behavioral Notion of Subtyping. *ACM Trans. Program. Lang. Syst.* 16, 6 (1994), 1811–1841. https://doi.org/10.1145/197320.197383

[26] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Catalin Hritcu, Exequiel Rivas, and Éric Tanter. 2019. Dijkstra monads for all. *PACMPL* 3, ICFP (2019), 104:1–104:29. https://doi.org/10.1145/3341708

[27] Bertrand Meyer. 1992. Applying "Design by Contract". *IEEE Computer* 25, 10 (1992), 40–51. https://doi.org/10.1109/2.161279

[28] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. 2008. Ynot: Dependent Types for Imperative Programs. In *ACM Sigplan Notices*, Vol. 43. ACM, 229–240.

[29] David Lorge Parnas. 1983. A Technique for Software Module Specification with Examples (Reprint). *Commun. ACM* 26, 1 (1983), 75–78. https://doi.org/10.1145/357980.358011

[30] François Pessaux. 2014. FoCaLiZe: inside an F-IDE. *arXiv preprint arXiv:1404.6607* (2014).

[31] Matthieu Sozeau. 2008. *Un environnement pour la programmation avec types dépendants*. Ph.D. Dissertation. Paris 11.

[32] Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings (Lecture Notes in Computer Science)*, Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar (Eds.), Vol. 5170. Springer, 278–293. https://doi.org/10.1007/978-3-540-71067-7_23

[33] Wouter Swierstra and Tim Baanen. 2019. A predicate transformer semantics for effects (functional pearl). *PACMPL* 3, ICFP (2019), 103:1–103:26. https://doi.org/10.1145/3341707

[34] Tarmo Uustalu. 2017. Container Combinatorics: Monads and Lax Monoidal Functors. In *Topics in Theoretical Computer Science - Second IFIP WG 1.8 International Conference, TTCS 2017, Tehran, Iran, September 12-14, 2017, Proceedings (Lecture Notes in Computer Science)*, Mohammad Reza Mousavi and Jirí Sgall (Eds.), Vol. 10608. Springer, 91–105. https://doi.org/10.1007/978-3-319-68953-1_8

[35] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction Trees: Representing Recursive and Impure Programs in Coq. In *The 47th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '20, New Orleans, USA - January 22 - 24, 2013*. ACM.

[36] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. 2013. Mtac: a monad for typed tactic programming in Coq. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 87–100. https://doi.org/10.1145/2500365.2500579