Mechanical Verification of Interactive Programs Specified by Use Cases

Guillaume Claret, Yann Régis-Gianas

Mechanical Verification of Interactive Programs specified by Use Cases

Guillaume Claret*, Yann Régis-Gianas*
* Univ Paris Diderot, Sorbonne Paris Cité,
PPS, UMR 7126 CNRS,
PiR2, INRIA Paris-Rocquencourt,
F-75205 Paris, France
http://guillaume.claret.me/ and http://yann.regis-gianas.org/

***Abstract*—Interactive programs, like user interfaces, are hard to formally specify and thus to prove correct. Some ideas coming from functional programming languages have been successful to improve the way we write safer programs, compared to traditional imperative languages, but these ideas mostly apply to code fragments without any inputs–outputs.**

Using the purely functional language Coq, we present a new technique to represent interactive programs and formally verify use cases using the Coq proof engine as a symbolic debugger. To this end we introduce the notion of *scenarios*, well-typed schema of interactions between an environment and a program. We design and certify a blog system as an illustration. Our approach generalizes unit-testing techniques and outlines a new method for mechanically assisted checking of effectful functional programs.

I. Introduction

Implementing and proving correct interactive programs is challenging. Indeed, interactive programs are hard to reason about because they communicate with an outer environment (the operating system, the network, the user, ...) which may be under-specified and non deterministic. Moreover, the communications between the program and the environment can happen at many points during the execution and may depend on previous interactions.

Many techniques have been developed to model, specify and prove correct interactive or concurrent programs[15]. For instance, process algebra and temporal logics are well understood *abstract models* for such programs. In these abstract models, some interesting behavioral properties can be automatically proved by model-checkers. Yet, these tools usually provide guarantees about the model of the program, not *its actual implementation.*

In another approach, called *software-proof co-design*, the specification and the verification of a program is not disconnected from its actual implementation. In that case, specifying, implementing and verifying are tightly interleaved in the software development process. This tight integration is possible within the Coq proof assistant which is both a programming language and an assisted prover. Yet, even if a realistic compiler for the C language has already been developed in Coq[12], using Coq as a general purpose programming language may be considered a *tour de force* especially because it is purely functional and only accepts total and terminating functions. Our objective is to alleviate the cost of certified programming (i) by importing monadic programming technique from the Haskell programming language to express effectful and partial computations ; (ii) by mechanizing mainstream software engineering techniques inside Coq to ease the migration from unverified to verified specifications.

In software development processes, use cases[10] are commonly used to specify and verify programs with inputs–outputs. A use case is basically a list of interaction steps between a program and an environment, in order to denote functional properties. A use case can be verified by manually running the program or by automated unit-testing. However, unit-testing methods are never complete as soon as there is an infinite set of possible values for the program inputs. We will show how to verify use cases on infinitely many inputs using the notion of *scenarios.*

A lot of ideas coming from functional programming languages changed the way we structure software today. Among these ideas are the use of composable higher-order functions as the main building block[9], referentially transparent expressions, algebraic data types, polymorphic[16] and dependent types[19]. Many programmers now use these ideas to increase the modularity and the correctness of their programs.

One of the limitations of most functional programming approaches is that they are often only effective on the pure (as "without inputs and outputs") fragments of a program. Other methodologies are required to write the impure (or interactive) parts (see for example [11]). For example, in the functional language Haskell, the impure computations are represented into a *monad*[23] using some special combinators. This monad isolates by typing the pure and impure computations, but offers limited ways to reason about the correctness of the inputs–outputs.

To extend and continue with the techniques developed in Haskell, we chose the purely functional language Coq to write and mechanically certify interactive programs. Coq is both a purely functional language and a theorem prover. The theory of Coq is based on the CIC[5] (the Calculus of Inductive Constructions type theory). It

features rich inductive and dependent types, terminating-by-construction expressions and the ability to mix proofs and computations. Its advanced module system allows to write large developments. Coq programs can be compiled to native code for maximum efficiency.

We develop a new method to write and certify interactive programs by formal use cases analysis using Coq. We illustrate our method by programming and certifying a *blog engine* with a web interface. The sources are available online on https://github.com/clarus/coq-chick-blog.

In this paper we will introduce and present:

- the notion of *computations* to express interactive computations in the purely functional language Coq;
- a semantics for the computations as the set of well-typed *runs*. A run describes both the trace, which can be built interactively, and the result of a computation;
- a new technique to express and prove functional properties over interactive programs. To this end, we introduce the concept of *scenarios* which are schema of well-typed interactions between a program and an environment. The scenarios are a mechanical formalization of the reasoning by use cases for verifying a program;
- a symbolic debugger for interactive programs, to assist the writing of scenarios and to spot bugs. This debugger relies on the existing tactics mode[6] of Coq, using this mode to explore the execution paths of an interactive computation;
- a blog system implemented and certified using our method. In particular, the user can login, add, edit or remove a post though a web interface. This blog system is compiled to an executable version using as an intermediate language the OCaml programming language (see http://ocaml.org/).

II. A challenge

Our challenge is to both develop and certify a blog engine in Coq. This challenge is interesting because a blog is a realistic example of a system with interactive user interactions.

Using this blog engine, the user is able to login, add, edit or delete a post. We save the posts to the file system, with one file per post. We take a very simple login system (no passwords) to concentrate on the architecture. The user interacts with the blog through HTTP requests.

We use an implementation of the HTTP protocol written in OCaml, a general purpose programming language. The blog itself is entirely written in Coq. We formally express and prove some non-functional properties and some functional properties reasoning by use cases. To generate a runnable executable of the blog, we compile the Coq code into OCaml code, and then to assembly code using the OCaml compiler.

Our trust base is composed of:

- the Coq system
- the compilation process from Coq to OCaml
- the OCaml compiler and runtime
- the implementation of HTTP in OCaml

III. Computations and runs

We will introduce the notions of *computations* and *runs* to represent and give a semantics to interactive programs in Coq.

A. Computations

A command is the value emitted during a call of a program to the outer environment. Let `Command.t` be a type of command and:

$$\text{answer} : \text{Command.t} \to \text{Type}$$

the dependent type of the answers to these commands. The set of interactive *computations* returning values of type A is represented by the type $\mathcal{C}\ A$, inductively defined in Coq by:

```
Inductive C (A : Type) : Type :=
| Ret : ∀ (x : A), C A
| Call : ∀ (c : Command.t), (answer c → C A) → C A.
```

This means that a computation can be either:

- a pure expression x of type A;
- a call to the environment with an argument c of type `Command.t` and a *handler* waiting for an answer of type `answer` c, dependent on the value of the command.

The role of a computation is to combine pure code fragments to form more complex programs interacting with the outer system.

Here is a computation printing the content of a file in the console:

```
1 Definition print_readme : C unit :=
2   Call (ReadFile "README") (fun text ⇒
3   match text with
4   | None ⇒ Ret ()
5   | Some text ⇒
6     Call (Log text) (fun _ ⇒
7     Ret ()))
8   end).
```

assuming the following relations:

$$\begin{aligned} \text{ReadFile} &: \text{string} \to \text{Command.t} \\ \text{Log} &: \text{string} \to \text{Command.t} \\ \text{answer ReadFile} &= \text{option string} \\ \text{answer Log} &= \text{unit} \end{aligned}$$

the unit type being a special type with just one value (), used for function which do not return any meaningful results.

We call an external procedure to read the file `README` on line 2. The variable `text` is set to the answer of the call, which is expected to be the content of the `README` file. If the content is `None` (in case of error), we return the unit value on line 4. If the content is some text, we print it

on the standard output using the command `Log` on line 6. Once the printing function has terminated, we return the unit value.

The values of the answers are left unspecified. We only assume there will be one answer of the right type for each call. We will see how to define execution traces and characterize the answers more precisely.

B. Runs

To reason about interactive programs we need to give a semantics to the computations. The semantics is defined by all the runs of a program. A *run* is an execution of a computation with explicit answers to the calls.

The type of the runs $\mathcal{R}$ is parametrized by a type A and a computation c of type $\mathcal{C}\,A$. We define it inductively in Coq, by symmetry with the definition of computations:

```
Inductive R (A : Type) : C A → Type :=
| RunRet : ∀ (x : A), R A (Ret x)
| RunCall : ∀ (c : Command.t) (a : answer c),
    ∀ {handler : answer c → C A}, (R A (handler a)) →
    R A (Call c handler).
```

A run can be either:

- a run of a `Ret` that carries the pure value x returned by a computation;
- a run of a `Call` of a command c that received an answer a of the corresponding type and a run of a handler applied to the answer a.

We do not explicitly write the *handler* terms since they are already in the definition of the computation and thus can be automatically inferred by Coq. Implicit parameters are declared into braces in Coq.

For the `print_readme` program, an example of run is:

```
Definition run_print_readme
  : Run unit print_readme :=
  RunCall (ReadFile "README") (Some "Blabla") (
  RunCall (Log "Blabla") () (
  RunRet ())).
```

This run tells us that the program will call a command `ReadFile "README"`. If we answer `Some "Blabla"`, then it will call the command `Log "Blabla"` to which the only possible answer, of type `unit`, is `()`. Then the program terminates without any other calls. We can see this run as a form of very weak specification of the program `print_readme`, describing its behavior only when we answer `Some "Blabla"` to the `ReadFile` operation.

A run describes both an execution trace of a computation and the result of its evaluation with this trace. Thus, we can extract the result of a run with the function eval:

```
Fixpoint eval {A : Type} {c : C A} (r : R A c) : A :=
  match r with
  | RunRet x ⇒ x
  | RunCall c a h r ⇒ eval r
  end.
```

We define `eval` by induction over a run. We recurse until we find the inner `Ret` and return its value. The `eval` function returns a value of type A as expected.

Similarly, we can extract the trace of a run:

```
Fixpoint trace {A : Type} {c : C A} (r : R A c)
  : list {c : Command.t & answer c} :=
  match r with
  | RunRet x ⇒ []
  | RunCall c a h r ⇒ (c, a) :: trace r
  end.
```

A trace is a list of dependent couples of commands and answers of the corresponding type. We recurse over a run, accumulating the command and the answer of each call in a list.

Notice that since Coq is a normalizing language, by construction, the trace is always a finite list and the evaluation of a run always terminates.

We have given a formal definition of interactive computations in Coq and defined the semantics of a computation as the set of its runs. We will show how we used the notion of computations to build a blog server.

IV. Programming the server

We will present the code of a blog engine, and how we used the notion of computations to implement the input/output operations.

A. The server handler

The main function of the blog server is:

$$\text{server} : \text{Path.t} \to \text{Cookies.t} \to \mathcal{C}\,\text{Response.t}$$

This function handles one request from the client. A request is a path (an URL, like `/login`) and the status of the client's cookies. A response is:

- a MIME type;
- a new set of cookies;
- a body, typically some HTML content.

The server function returns a *computation* of response since it does calls to the system. The state of the blog is saved on the file system and accessed through system calls. In total, the blog system itself is made of 786 lines of Coq code.

The server function is pure, expect for the uninterpreted input/output operations. In particular, this function is deterministic, cannot return any exceptions and always terminates by construction. This is given to us for free, thanks to the strict type-system of Coq.

The path of the request, initially a string URL, is parsed by a pure Coq function to the sum type *Path.t*. A sum type is a union of different types, each introduced by a constructor. The constructors of the *Path.t* type are given in the figure 1.

This explicit sum type also describes the web API of the blog application. The `NotFound` and `WrongArguments` constructors are for ill-formed requests. `Static` retrieves

Fig. 1. Constructors of the *Path.t* type.

Constructor	Arguments	Root path
`NotFound`		
`WrongArguments`		
`Static`	list string	`/static`
`Index`		`/`
`Login`		`/login`
`Logout`		`/logout`
`PostAdd`		`/posts/add`
`PostDoAdd`	string × date	`/posts/do_add`
`PostEdit`	string	`/posts/edit`
`PostDoEdit`	string × string	`/posts/do_edit`
`PostDoDelete`	string	`/posts/do_delete`
`PostShow`	string	`/posts/show`

static content such as CSS files. `Index` shows the main page. You use `Login` and `Logout` to login and logout (there is no passwords or user names). `PostAdd` shows the form to add a post, `PostDoAdd` effectively add a post. So do `PostEdit` and `PostDoEdit` to edit a post. `PostDoDelete` removes a post. Finally, `PostShow` shows the content of a post.

B. Edit a post

We will see in details how we implemented the handler for the `PostDoEdit` requests. This is the code extract of the function `post_do_edit`, which generates a response for a request of the form:

/`posts/do_edit`/*url*?`content` =*content*

```
1  Definition post_do_edit (is_logged : bool)
2    (url : string) (content : string) : C Response.t :=
3    if negb is_logged then
4      ret Response.Forbidden
5    else
6      let! is_success : bool := fun k ⇒
7        let! header := Helpers.header url in
8        match header with
9        | None ⇒ k false
10       | Some header ⇒
11         let file_name := posts_directory ++
12           Post.Header.file_name header in
13         call! is_success :=
14           UpdateFile file_name content in
15         k is_success
16       end in
17     ret (Response.PostDoEdit url is_success).
```

We check on line 3 if the user is logged. If so, we call the function `Helpers.header` on line 7 which gets the meta-data of a post from the file system. Its type is:

$$\forall\, A,\ \text{string} \to (\text{option}\ \text{Post.Header.t} \to \mathcal{C}\ A) \to \mathcal{C}\ A$$

Fig. 2. Calls we use in the blog.

Command	Arguments	Answer
`ReadFile`	string	option string
`UpdateFile`	string × string	bool
`DeleteFile`	string	bool
`ListPosts`	string	option (list header)
`Log`	string	unit

This function `Helpers.header` waits for a continuation, a function returning the next computation. The notation `let!` on line 7 is just a syntactic sugar to call a function with a continuation:

$$\texttt{let!}\ x := e_1\ \texttt{in}\ e_2 \quad \Longleftrightarrow \quad e_1\,(\lambda x.\, e_2)$$

We program by continuations to compose computations. Since the monads can generalize the continuation-passing style[17], we could have added the operator `bind` operator:

$$\texttt{Bind} : \forall\, A\, B,\ \mathcal{C}\ A \to (A \to \mathcal{C}\ B) \to \mathcal{C}\ B$$

but we preferred to keep our number of primitives as small as possible. We believe it simplifies the reasoning over the runs, but we are opened to discussions on this point.

If the header is not available, we return `false` for the success status to the continuation k on line 9. Otherwise we call the command `UpdateFile` on line 14 to update the content of the post on the file system. The notation `call!` on line 13 is a syntactic sugar for a call:

$$\texttt{call!}\ x := c\ \texttt{in}\ e \quad \Longleftrightarrow \quad \texttt{Call}\ c\ (\lambda x.\, e)$$

The list of the calls we use is given on figure 2.

Finally, we return a page `PostDoEdit` on line 17 with a link to the original post and the success status of the update.

The Response.t type is a sum type, with one constructor per kind of page. A purely functional pretty-printer then renders the corresponding HTML content. The pretty-printer is not verified.

We have seen how to program a blog engine in a type-safe manner in Coq using the notion of computations. In the next section, we will see how we verified this blog system.

V. Specifying the server

The type-system of Coq already provides some safety properties on the computations, like the termination. We will go further, proving non-functional and functional properties using our notion of scenarios.

A. Scenario

A *scenario* is a set of *runs* of a *computation*. A scenario whose definition is accepted by Coq is a set of runs validated by its computation. The idea is to specify the behavior of the blog reasoning by use cases, verifying functional requirements.

A scenario is usually described as a family of runs parametrized by some user inputs. To be relevant, the use case or the expression of the scenario should be as clear as possible. An example of an informal use case is the following:

1) I add a new post as an authenticated user
2) I edit this post with some content c
3) I show this post and check that this is the content c I edited

We formally verified this use case by writing a well-formed scenario. To keep the explanations short, we will study the simpler use case of the index page service:

1) I connect to the index page URL
2) the blog calls the file system to list the available posts
   - in case of error, a log message is printed on the server console
3) the index page is displayed with the list of posts

B. Symbolic debugger

To write scenarios we leverage the tactics mode of Coq. The tactics mode is normally used to prove theorems reasoning step by step. By using this mode to define scenarios, we get a kind of interactive debugger for computations: it is possible to evaluate a computation stepping through each call, like we would iterate into the reasoning steps of a theorem. At each call an explicit answer must be provided to go to the next call. The trace of the user interactions with the debugger is then a *run* of a computation.

More important, this debugger is symbolic: Coq being designed to manipulate symbolic expressions, we do not need to instantiate variables with concrete values. So the interactions with the debugger can actually describe sets of *runs*, that is to say *scenarios*.

The scenario of the index page when there are no errors is the following:

```
1  Definition index_ok (cookies : Cookies.t)
2    (headers : list Header.t)
3    : Run.t (Main.server Path.Index cookies).
4    simpl.
7    apply (RunCall (ListPosts _) (Some headers)).
8    apply (RunRet (Response.Index
9      (Cookies.is_logged cookies)
10     headers)).
11 Defined.
```

This scenario describes a run of the server handler for each user cookie and list of post headers. We do not need to execute these runs: because `index_ok` is well-typed, we know they are all correct. They do just one call `ListPosts` and display the list of posts. This is a big advantage compared to unit-testing which would requires to run the program on each possible input, what is exhaustively impossible here since there are infinitely many inputs.

To construct this scenario using the debugger (the tactics mode), we type the first three lines. The Coq interpreter replies:

```
1 subgoals
cookies : Cookies.t
headers : list Header.t
______________________________________(1/1)
Run.t (Main.server Path.Index cookies)
```

This means that we have two symbolic parameters, cookies and headers, and aim to construct a run of the server handler applied to the index path and the cookies. We enter the `simpl` command on line 4 to partially evaluate the computation. Coq will use the fact that `Path.Index` is a concrete value. We get:

```
1 subgoals
cookies : Cookies.t
headers : list Header.t
______________________________________(1/1)
Run.t (Main.Controller.index
  (Cookies.is_logged cookies))
```

We can guess what will be the next call unfolding the `Main.Controller.index` definition with the command `unfold` or looking directly at its source code on line 3 on figure 3.

```
1  Definition index (is_logged : bool)
2    : C.t Response.t :=
3    call! posts := ListPosts posts_dir in
4    match posts with
5    | None =>
6      do_call! Log ("Cannot open the " ++
7        posts_dir ++ " directory.") in
8      ret (Response.Index is_logged [])
9    | Some posts =>
10     ret (Response.Index is_logged posts)
11   end.
```

Fig. 3. Source code of the index function

We guess it is `ListPosts` to some folder, to which we answer `Some headers` on line 7:

```
apply (RunCall (ListPosts _) (Some headers)).
```

The Coq system validates our guess, unifying modulo evaluation the computation:

```
Main.Controller.index (Cookies.is_logged cookies)
```

with a computation of the form:

```
Call (ListPosts ...) (fun a ⇒ ...)
```

We continue the evaluation of the computation by one step and get to:

```
1 subgoals
cookies : Cookies.t
headers : list Header.t
______________________________________(1/1)
Run.t (C.Ret (Response.Index
  (Cookies.is_logged cookies)
  headers))
```

Since we are on a `Ret` expression, the evaluation is terminated and we can conclude by the line 8, which explicitly states the expected result. In particular, we require the response to be the index page and to include the list of `headers`.

Likewise, we have defined a scenario for the index page with errors from the file system, by answering `None` to the call `ListPosts`. We have also verified the add, edit and show scenario.

In our experience, defining a scenario is similar to writing a unit-test. Indeed, a unit-test for an interactive program is basically a trace of all the responses of the environment to the program. This is exactly the same for a scenario, whereas the responses can be symbolic (representing an infinite set of possible values) and not only concrete (representing a single value).

C. Non-functional properties

We also verify some non-functional properties. Even if this not the main goal of the article, it shows that the notion of computations is not only useful for the certification of use cases.

For example, we prove than an unauthenticated user cannot make a request which generates calls modifying the file system. To do so, we define a predicate:

$$\text{is\_read} : \text{Command.t} \to \text{bool}$$

to check that a command does not modify the file system. By induction over a computation, we define what is a computation free of write operations:

$$\begin{array}{l}
\texttt{Inductive}\ \text{read_only}\ \{A : \text{Type}\} : \mathcal{C}\ A \to \text{Prop} := \\
\mid\ \texttt{RoRet} : \forall\, x : A,\ \text{read_only}\ (\texttt{Ret}\ x) \\
\mid\ \texttt{RoCall} : \\
\quad \forall\, (c : \text{Command.t})\ (\mathit{handler} : \text{answer}\, c \to \mathcal{C}\ A), \\
\quad \text{is_read}\, c = \text{true} \to \\
\quad (\forall\, (a : \text{answer}\, c),\ \text{read_only}\ (\mathit{handler}\, a)) \to \\
\quad \text{read_only}\ (\texttt{Call}\, c\, \mathit{handler}).
\end{array}$$

By disjunction on the path, we show in Coq that the predicate read_only is valid for any computation handling a request from an unauthenticated user:

$$\begin{array}{l}
\forall\, (\mathit{path} : \text{Path.t}), \\
\quad \text{read_only}\ (\text{Main.server}\, \mathit{path}\, \text{Cookies.LoggedOut})
\end{array}$$

which proves our claim.

We have seen how to show both functional and non-functional properties over a computation. We can prove functional properties in an interactive way using Coq as a symbolic debugger.

VI. Compiling the server

We will explain how we compiled the *computation* of the blog system to an executable program, using an automatic translation to the OCaml language.

The Coq programming language is a purely functional language. We can represent the input/output operations but we cannot execute them. To do so, we use the *extraction* mechanism[13], which compiles Coq programs to the impure programming language OCaml.

We introduce some uninterpreted constants and extract them to specific OCaml impure expressions. For example:

```
Parameter printl : String.t -> t unit.
Extract Constant printl => "Lwt_io.printl".
```

declares the Coq constant `printl` and associates it to the OCaml function `Lwt_io.printl`, which prints a line on the standard output. After extraction, this function will effectively display a line on the screen. As mentioned earlier, this compilation process is part of the trusted base.

We link the extracted code to the OCaml library CoHTTP to handle the HTTP protocol. CoHTTP creates the main program loop, waiting for HTTP requests. The event-based concurrency model is managed by the Lwt library[22], a cooperative lightweight threads library for OCaml.

VII. Related work

Effectful functional programming have been intensively studied by the Haskell community, and large real programs have been written using this functional language. The main technique popularized by Haskell is the IO monad to express impure computations and to verify by typing the isolation of pure and impure expressions. Simon Peyton Jones gave a nice summary of the IO monad and other techniques[11]. We wanted to build upon this experience exploring new solutions, thanks to the more powerful type-system of Coq featuring dependent-types and propositional types.

The Ynot project[18] studied the use of a parametrized monad to represent and reason about impure computations in Coq. This project focused more on the imperative and low-level memory management, using Hoare-logic[8] with pre- and post-conditions together with separation logic[21]. They explored extensions to reason about inputs–outputs and have written and mechanically certified a web application[14]. Unlike our work, their application is specified by an invariant over the execution trace of the program. They do not study the verification of functional properties, or use cases.

The algebraic effects and handlers[20], a generic framework to represent effects in a compositional way in purely functional languages, led to a lot of research about proven safe effectful programs. This framework is more powerful than ours because it can represent many different kinds of effects, like non-determinism, exceptions, or states, and can combine them in a generic way. This power has a cost: the algebraic effects are also more complex to define and understand. It could be interesting to see if our notion of *computations* can be viewed as a particular case of algebraic effect.

The dependently typed programming language Idris[1] proposes an implementation of algebraic effects[2]. Edwin

Brady and Simon Fowler show how to specify the rules of a game or a web protocol and how to verify by typing their implementations in Idris (see [3] and [7]). These works mostly focus on expressing invariants and building the right primitives to write correct-by-construction programs. By contrast, we focused on specifications by use cases and on a tool, the symbolic debugger, to express these cases interactively.

Ur/Web[4] is a functional programming language and a platform for the web development made by Adam Chlipala. We can cite the BazQux Reader, a commercial RSS feed reader, as a successful application of the Ur/Web platform. The language Ur does not feature full dependent types, but can generate formally valid and unexploitable web pages and SQL requests thanks to its rich type system and its integrated platform. Combining our notion of *scenarios* with the ideas of the Ur/Web platform could be an interesting subject.

VIII. Conclusion

We have presented the notions of *computations* and *runs* to represent and to give a semantics to interactive programs in the purely functional language Coq. We have shown how to define and prove non-functional properties on a computation using predicates inductively defined on the combinators of the computations. We have introduced the notion of *scenarios* to denote and verify functional properties or use cases of a program. The usage of the tactic mode of Coq provides a symbolic debugger to write these scenarios interactively. To illustrate this approach, we developed and certified a blog system in Coq.

We would like to extend our techniques to a wider class of programs. In particular, we want to support asynchronous calls and concurrent computations. To add concurrency, we could use a *fork* primitive or an actor model. As an application, this would allow us to certify together the blog system, its HTTP front-end and its database or file system back-end, even in the case of concurrent user requests.

References

[1] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013.

[2] Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 133–144. ACM, 2013.

[3] Edwin Brady. Resource-dependent algebraic effects. In Jurriaan Hage and Jay McCarthy, editors, *Trends in Functional Programming*, Lecture Notes in Computer Science, pages 18–33. Springer International Publishing, 2015.

[4] Adam Chlipala. Ur/web: A simple model for programming the web. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 153–165. ACM, 2015.

[5] T. Coquand and Gérard Huet. The calculus of constructions. Technical Report RR-0530, May 1986.

[6] David Delahaye. A tactic language for the system coq. In Michel Parigot and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95. Springer, 2000.

[7] Simon Fowler and Edwin Brady. Dependent types for safe and secure web programming. In Rinus Plasmeijer, editor, *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages, Nijmegen, The Netherlands, August 28-30, 2013*, page 49. ACM, 2013.

[8] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[9] P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[10] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-oriented software engineering - a use case driven approach*. Addison-Wesley, 1992.

[11] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In *Engineering theories of software construction*, pages 47–96. Press, 2001.

[12] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[13] Pierre Letouzey. Extraction in coq: An overview. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings*, volume 5028 of *Lecture Notes in Computer Science*, pages 359–369. Springer, 2008.

[14] Gregory Malecha, Greg Morrisett, and Ryan Wisnesky. Trace-based verification of imperative programs with i/o. *J. Symb. Comput.*, 46(2):95–118, February 2011.

[15] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[16] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[17] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 14–23. IEEE Computer Society, 1989.

[18] Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Reasoning with the awkward squad. In *In ACM SIGPLAN International Conference on Functional Programming*, 2008.

[19] Ulf Norell. Dependently typed programming in agda. In Andrew Kennedy and Amal Ahmed, editors, *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, pages 1–2. ACM, 2009.

[20] Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009.

[21] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002.

[22] Jérôme Vouillon. Lwt: a cooperative thread library. In Eijiro Sumii, editor, *Proceedings of the ACM Workshop on ML, 2008, Victoria, BC, Canada, September 21, 2008*, pages 3–12. ACM, 2008.

[23] Philip Wadler. The essence of functional programming. In Ravi Sethi, editor, *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, pages 1–14. ACM Press, 1992.