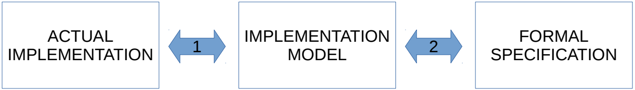
MECHANICAL VERIFICATION OF INTERACTIVE PROGRAMS SPECIFIED BY USE CASES

Guillaume Claret (`guillaume.claret@inria.fr`)
Yann Régis-Gianas (`yrg@pps.univ-paris-diderot.fr`)

INRIA $\pi.R^2$ – CNRS PPS – Université Paris Diderot

How to **mechanically prove** that
a program respects its **formal specification**?

Software certification: a model-centric approach


ACTUAL IMPLEMENTATION

1

IMPLEMENTATION MODEL

2

FORMAL SPECIFICATION

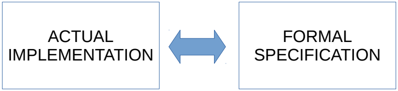
Languages

- Specification: Temporal logic, Hoare triples, ...
- Implementation model : Process calculus, Labelled transition systems, ...
- Actual implementation: C, C++, Ada, Java, ...

Tools and Techniques

- For 2 : model-checking, deductive reasoning, abstract interpretation, ...
- For 1 : refinement, certified encoding, faith, ...

Software certification: a language-centric approach



ACTUAL IMPLEMENTATION

FORMAL SPECIFICATION

Languages

- Specification: Types as a universal language.
- Implementation: High-level programming languages with formal semantics.

Tools and Techniques

- Curry-Howard correspondence:
  - a type is a formula ;
  - a program of that type is a proof of that formula.
- Software-Proof Co-Design.

The Coq proof assistant

http://coq.inria.fr

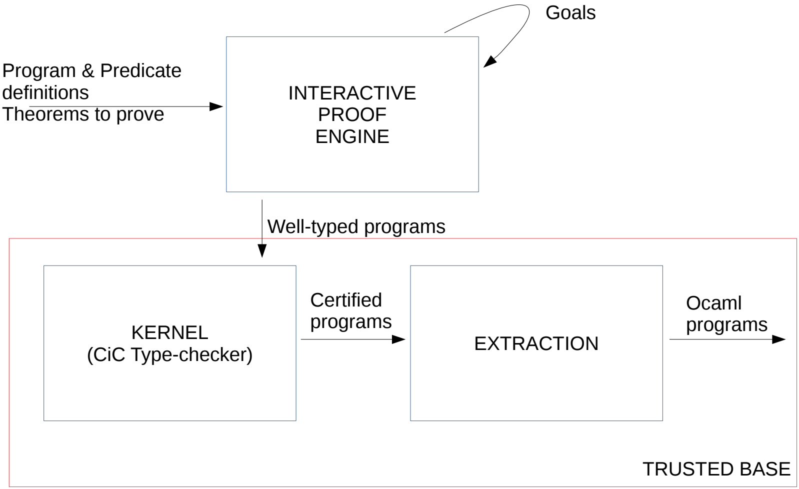
In a nutshell

- Almost 30 years of research in Logic and Computer Science.
- The Calculus of Inductive Constructions:
  Both a programming language and a logic.
- CiC enjoys the Curry-Howard correspondence.
- A very expressive logic.
- A high-level functional programming language.

Achievements

- Mathematical side: four colors, Feit Thompson, . . .
- Computer science side: CompCert, . . .
- ACM awards.

The Coq proof assistant : De Bruijn architecture at work



Goals

Program & Predicate definitions
Theorems to prove

INTERACTIVE PROOF ENGINE

Well-typed programs

KERNEL (CiC Type-checker)

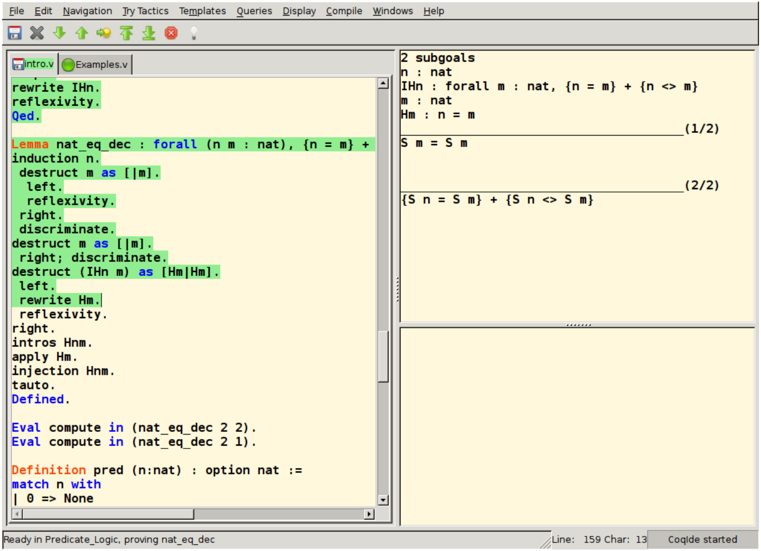
Certified programs

EXTRACTION

Ocaml programs

TRUSTED BASE

The Coq proof assistant



```
rewrite IHn.
reflexivity.
Qed.

Lemma nat_eq_dec : forall (n m : nat), {n = m} +
induction n.
 destruct m as [|m].
  left.
  reflexivity.
 right.
 discriminate.
destruct m as [|m].
 right; discriminate.
destruct (IHn m) as [Hm|Hm].
 left.
 rewrite Hm.
 reflexivity.
right.
intros Hnm.
apply Hm.
injection Hnm.
tauto.
Defined.

Eval compute in (nat_eq_dec 2 2).
Eval compute in (nat_eq_dec 2 1).

Definition pred (n:nat) : option nat :=
match n with
| 0 => None
```

```
2 subgoals
n : nat
IHn : forall m : nat, {n = m} + {n <> m}
m : nat
Hm : n = m
______________________________________(1/2)
S m = S m
______________________________________(2/2)
{S n = S m} + {S n <> S m}
```

```
Ready in Predicate_Logic, proving nat_eq_dec
Line:  159 Char:  13
CoqIde started
```

This talk

How to write and prove correct
interactive programs
within the Coq proof assistant?

Three questions

1. How to represent interactive programs in Coq?
2. What is the semantics of these programs?
3. How to prove properties about the behavior of these programs?

How to represent interactive programs in Coq?

Coq is a purely functional programming language

Key programming mechanisms

- Higher-order functions
- Pattern matching over inductively-defined data
- Dependent types
- Module system and type classes.

Restrictions (because it is also a logic)

- Effect-free: no assignment, no input-output, . . .
- Normalizing : all computations must terminate.

Coq is a purely functional programming language

Key programming mechanisms

- Higher-order functions
- Pattern matching over inductively-defined data
- Dependent types
- Module system and type classes.

Restrictions (because it is also a logic)

- Effect-free: no assignment, no input-output, . . .
- Normalizing : all computations must terminate.

Interactive programs do not terminate and perform I/O . . .
Are they out of Coq's scope?

Coq can represent interactive computations

An old wisdom from Haskell programmers:

> Even if a purely functional language cannot do effects,
> it can **represent** them thanks to **monads**.

The trick (to be efficient):

> The compiler **can optimize** their interpretation
> **using actual effects**.

A type for commands and answers

Definitions

Assume that `Command.t` is the type for commands and that there exists a dependent type `answer` of type:

$$\texttt{Command.t} \to \mathsf{Type}$$

representing the type of the environment answer to a command.

Examples

$$\begin{array}{rcl}
\mathrm{ReadFile} & : & \mathrm{string} \to \mathrm{Command.t} \\
\mathrm{Log} & : & \mathrm{string} \to \mathrm{Command.t} \\
\mathrm{answer}\,\mathrm{ReadFile} & = & \mathrm{option}\,\mathrm{string} \\
\mathrm{answer}\,\mathrm{Log} & = & \mathrm{unit}
\end{array}$$

A representation of interactive computations

The type of interactive computation $\mathcal{C}$ producing a value of type A is:

$$\begin{array}{l}
\texttt{Inductive}\ \mathcal{C}\ (A : \mathrm{Type}) : \mathrm{Type} := \\
\mid \texttt{Ret} : \forall\,(x : A),\ \mathcal{C}\ A \\
\mid \texttt{Call} : \forall\,(c : \mathrm{Command.t}),\ (\mathrm{answer}\ c \to \mathcal{C}\ A) \to \mathcal{C}\ A.
\end{array}$$

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.

A representation of interactive computations

Remarks

- A computation is nothing but a **well-typed Abstract Syntax Tree**.
- A computation combines pure code fragments to form more complex programs interacting with the outer system.
- Strictly speaking, computations are not a monad but an embedded DSL (close the algebraic effects of the IDRIS programming language).

Example

```
Definition print_readme : 𝒞 unit :=
  Call (ReadFile "README") (fun text ⇒
  match text with
  | None ⇒ Ret ()
  | Some text ⇒
    Call (Log text) (fun _ ⇒
    Ret ())
  end).
```

Syntactic sugar

$$\begin{aligned} \texttt{call!}\ x := c\ \texttt{in}\ e \quad &\Longleftrightarrow \quad \texttt{Call}\ c\ (\lambda x.\, e) \\ \texttt{ret}\ e \quad &\Longleftrightarrow \quad \texttt{Ret}\ e \end{aligned}$$

Example

```
Definition print_readme : 𝒞 unit :=
  call! text := ReadFile "README" in
  match text with
  | None ⇒ ret ()
  | Some text ⇒
    call! r := Log text in
    ret ()
  end.
```

What is the semantics of these programs?

Semantics by completion

Computations are incomplete

In general, a computation of type $\mathcal{C}\,A$ cannot produce a value of type A because it lacks the answers of the environment to the commands performed by the program.

Semantics by completion

Computations are incomplete

In general, a computation of type $\mathcal{C}\,A$ cannot produce a value of type A because it lacks the answers of the environment to the commands performed by the program.

How should we **complete a computation**
with these pieces of information?

A dependent type to represent the environment answers

The type $\mathcal{R}\ A\ c$ is the type for the *run*s of the computation c of type $\mathcal{C} A$:

$$\begin{array}{l}
\texttt{Inductive}\ \mathcal{R}\ (A : \mathrm{Type}) : \mathcal{C}\,A \to \mathrm{Type} := \\
\mid \texttt{RunRet} : \forall\,(x : A),\ \mathcal{R}\,A\,(\texttt{Ret}\,x) \\
\mid \texttt{RunCall} : \forall\,(c : \mathrm{Command.t})\,(a : \mathrm{answer}\ c), \\
\quad \forall\,\{\mathit{handler} : \mathrm{answer}\ c \to \mathcal{C}\ A\},\ (\mathcal{R}\,A\,(\mathit{handler}\ a)) \to \\
\quad \mathcal{R}\,A\,(\texttt{Call}\ c\ \mathit{handler}).
\end{array}$$

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.

Example

```
Definition run_print_readme : Run unit print_readme :=
  RunCall (ReadFile "README") (Some "Content of the file") (
  RunCall (Log "Content of the file") () (
  RunRet ())).
```

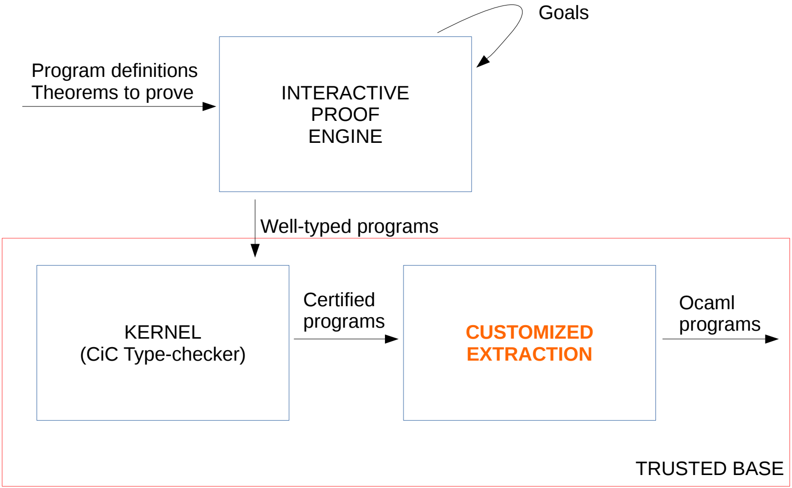
Big-step semantics

```
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.
```

Trace-based semantics

```
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.
```

Compilation



Goals

Program definitions
Theorems to prove

INTERACTIVE
PROOF
ENGINE

Well-typed programs

KERNEL
(CiC Type-checker)

Certified
programs

**CUSTOMIZED
EXTRACTION**

Ocaml
programs

TRUSTED BASE

How to prove properties about these programs?

Theorems based on the semantics

Standard correctness properties

- Given a computation c, any **extensional** property P about the final result of the program can be stated as soon as we do a universal quantification over runs:

$$\forall (r : \mathcal{R}\, A\, \mathsf{c}), P(\mathit{eval}\ r)$$

- Any **intentional** property P about the interaction between the program and its environment can also be stated:

$$\forall (r : \mathcal{R}\, A\, \mathsf{c}), P(\mathit{trace}\ r)$$

Yet, in the case of an **interactive** program, specifications are more naturally written as the union of use-case **scenarios**.

Scenarios

Definition

A **scenario** is a (possibly infinite) family of *runs* parameterized by user inputs.

Scenario as specification

A well-typed scenario is a valid specification for the interaction between the environment (which includes the user) and the program.

Scenarios

Definition

A **scenario** is a (possibly infinite) family of *runs* parameterized by user inputs.

Scenario as specification

A well-typed scenario is a valid specification for the interaction between the environment (which includes the user) and the program.

Scenarios are formal representations for use-cases.

Type-checking a scenario validates the implementation with respect to the use-cases it represents.

Example

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

Definition run_print_readme_ko : Run unit print_readme :=
  RunCall (ReadFile "README") None (
  RunRet ())).
```

Case study: Development of a blog engine

Is that approach realistic? (Work in Progress)

A small experiment

- We develop a blog engine, *i.e.* a server of type:

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

- 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.
- 786 lines of Coq
- By construction: deterministic, no exceptions, always terminates.

The type for paths

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`

The type for commands

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

Interactive constructions of scenarios

- We wrote scenarios for all the possible requests to the blog engine.
- It was almost impossible to correctly write formal scenarios manually: there are too many details and cases to consider.
- Hopefully, scenarios can be written **interactively** with the help of the interactive proof engine of Coq.
- The type system makes sure that no case is missed.

Interactive constructions of scenarios

- We wrote scenarios for all the possible requests to the blog engine.
- It was almost impossible to correctly write formal scenarios manually: there are too many details and cases to consider.
- Hopefully, scenarios can be written **interactively** with the help of the interactive proof engine of Coq.
- The type system makes sure that no case is missed.

The interactive proof engine of Coq is here used as a **symbolic debugger**.

Example

Consider the following use-case:

1. The user connects to the index page URL.
2. The blog calls the file system to list the available posts.
3. In case of error, a log message is printed on the server console.
4. Otherwise, the index page is displayed with the list of posts.

Example

This amounts to find a proof for:

```
Definition index_ok (cookies : Cookies.t)(headers : list Header.t)
    : Run.t (Main.server Path.Index cookies).
```

Example

After entering these lines to Coq, a goal is produced:

```
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, $\mathrm{cookies}$ and $\mathrm{headers}$, and aim to construct a run of the server handler applied to the index path and the cookies. We enter the `simpl` command to partially evaluate the computation using the fact that `Path.Index` is a concrete value.

Example

```
Definition index_ok (cookies : Cookies.t)
  (headers : list Header.t)
  : Run.t (Main.server Path.Index cookies).
  simpl.
```

Example

We get:

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

The next call must be `ListPosts` to some folder, to which we answer `Some headers`:

`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 ⇒ ...)`

Example

```
Definition index_ok (cookies : Cookies.t)
  (headers : list Header.t)
  : Run.t (Main.server Path.Index cookies).
  simpl.
  apply (RunCall (ListPosts _) (Some headers)).
```

Example

The next subgoal is:

```
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 stating the expected result: we require the response to be the index page and to include the list of `headers`.

Example

```
Definition index_ok (cookies : Cookies.t)
  (headers : list Header.t)
  : Run.t (Main.server Path.Index cookies).
  simpl.
  apply (RunCall (ListPosts _) (Some headers)).
  apply (RunRet (Response.Index
    (Cookies.is_logged cookies)
    headers)).
Defined.
```

Conclusion and future work

Ideas to take home

- Interactive programs can be developed, specified and certified within Coq.
- Scenarios, *i.e.* symbolic use-cases, can be built interactively.
- Type-checking ensures that programs interact well.

Future work

Research agenda

- A theory of use-cases to mechanically prove that:
  - a use-case refines or extends another use-case ;
  - a set of use-cases covers all the behaviors of a program.
- A temporal logic in CiC and related proof system on computations.
- Concurrency primitives and a model-checker.
- Confront this technique with larger software developments.

More about this project…

- `https://github.com/clarus/io`
- `http://coq-blog.clarus.me/`

Thank you for your attention!
Any question?