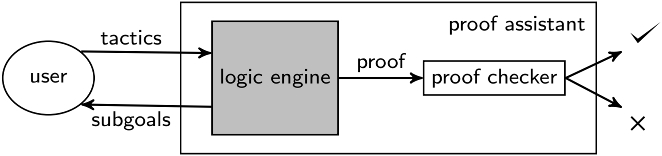
The Coq Proof Assistant: Introduction and Basics

Karl Palmskog

01/25/18

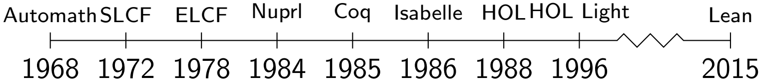
Verification Using Proof Assistants

1. encode definitions in (higher-order) formalism
2. prove propositions **interactively** using powerful **tactics**
3. check soundness of every low-level step



examples: Coq, HOL4, HOL Light, Isabelle/HOL, Lean, Nuprl, ...

Proof Assistants In Perspective



- in use for over 40 years, mostly in academia
- more trustworthy than testing, model checking, ...
- expensive to apply (expertise, time, opportunity cost, ...)

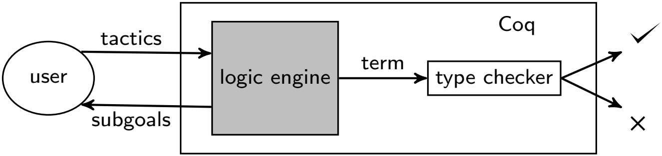
Some Large-Scale Proof Assistant Projects

Project	Year	Assistant	LOC
4-Color Theorem	2005	Coq	60k
Odd Order Theorem	2012	Coq	150k
Kepler Conjecture	2015	HOL Light	500k
CompCert	2009	Coq	40k
seL4	2009	Isabelle/HOL	200k
Verdi Raft	2016	Coq	50k

The Coq Proof Assistant: `http://coq.inria.fr`

Coq can be viewed as consisting of two things:

1. a simple, pure and expressive typed programming language
2. a set of tools for stating logical properties and proving them



Coq in Theory

- descendant of Martin-Löf's intuitionistic type theory
- follows the De Bruijn principle of small, hand-verified core
- uses tactical proving like Milner's LCF
- strongly normalizing type system
- LEM and extensional equality of functions do not hold!

Coq in Practice

- implemented in OCaml
- GNU LGPL
- in development for 20+ years by 40+ people
- extensive use in academia for mathematics and software
- many interfaces (emacs/ProofGeneral, CoqIDE, Coqoon, ...)

Why Modelling and Verification in Coq?

- functions can be extracted to OCaml/Haskell/Scheme
- no restriction to bounded structures (cf. model checking)
- higher-order structures possible anywhere (maps, sets, ...)
- embed your favorite programming language and its semantics
- validation of theories possible by proving “meta-level” lemmas
- lots of libraries, papers, documentation

Why Not Modelling and Verification in Coq?

- modest built-in proof automation
- encoding a theory sometimes requires foundational knowledge
- high symbol pushing overhead for new domains
- inappropriate encodings a big issue

Coq Theory Intuitions

- a type is a term and a term is a type (repeat ten times!)
- "p is a proof P" means that term p has type P
- exists proof of P precisely when P is inhabited by some term

Coq Theory Intuitions, Continued

- there is no proof of $\bot$
- there is precisely one proof of $\top$
- a proof of $P \wedge Q$ is a term (p, q) where
  - p is a proof of P
  - q is a proof of Q
- a proof of $P \vee Q$ is either $(\texttt{left}, p)$ or $(\texttt{right}, q)$
- a proof of $P \rightarrow Q$ is a function that converts p into q

Coq Theory Intuitions, Continued

- a proof of $\forall x : X,\ \phi(x)$ is a function that converts any x of type X into a proof of $\phi(x)$
- a proof of $\exists x : X, \phi(x)$ is a term (a, p) where a has type X and p is a proof of $\phi(a)$
- a proof $\neg P$ is a function that converts p into a proof of $\bot$

Interaction with Coq

- LCF proof assistants required use of read-eval-print loop
- all interaction was synchronous
- Coq still provides such an interface

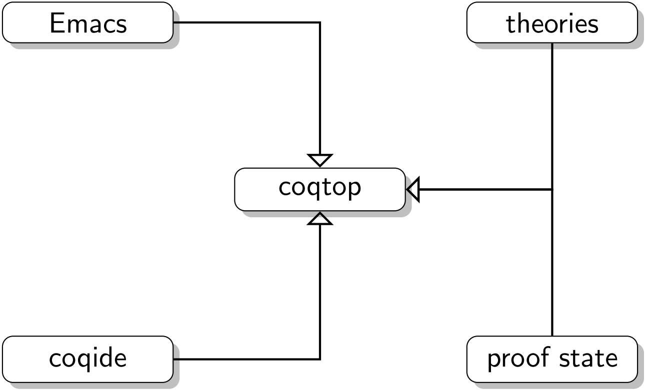
```
# coqtop
Welcome to Coq 8.7.1 (December 2017)

Coq < Lemma alt_exists : forall l1 l2, exists l3, alt l1 l2 l3.
1 subgoal

  ============================
  forall l1 l2 : list nat, exists l3 : list nat, alt l1 l2 l3

alt_exists < induction l1; intros; destruct l2.
```

Coq Interaction Overview



Function Definitions in Gallina

```
Require Import List.
Import ListNotations.

Fixpoint alternate (l1 l2 : list nat) : list nat :=
match l1 with
| [] ⇒ l2
| h1 :: t1 ⇒
  match l2 with
  | [] ⇒ h1 :: t1
  | h2 :: t2 ⇒ h1 :: h2 :: alternate t1 t2
  end
end.
```

Computing Inside Coq With Vernacular Commands

```
Eval compute in alternate [1] [2].
(*     = [1; 2]
    : list nat *)

Eval compute in alternate [1; 3; 5] [2; 4; 6].
(*     = [1; 2; 3; 4; 5; 6]
    : list nat *)
```

Function Specifications in Gallina (compare Prolog)

```
Inductive alt : list nat → list nat → list nat → Prop :=
| alt_nil :
   forall l, alt [] l l
| alt_step :
   forall a l t1 t2,
   alt l t1 t2 →
   alt (a :: t1) l (a :: t2).
```

Proving Simple Properties Using Ltac tactics

```
Lemma alt_one : alt [] [1] [1].
Proof.
(* 1 subgoal, subgoal 1 (ID 30)

  ============================
  alt [] [1] [1] *)
apply alt_nil.
(* No more subgoals. *)
Qed.
(* alt_one is defined *)
```

Proving Simple Properties Using Ltac tactics

```
Lemma alt_123456 : alt [1; 3; 5] [2; 4; 6] [1; 2; 3; 4; 5; 6].
Proof.
(* 1 subgoal, subgoal 1 (ID 46)

  ============================
  alt [1; 3; 5] [2; 4; 6] [1; 2; 3; 4; 5; 6] *)
apply alt_step.
(* 1 subgoal, subgoal 1 (ID 47)

  ============================
  alt [2; 4; 6] [3; 5] [2; 3; 4; 5; 6] *)
apply alt_step.
apply alt_step.
apply alt_step.
apply alt_step.
apply alt_step.
apply alt_nil.
Qed.
```

Proving Properties Using Induction

```
Lemma alt_alternate :
  forall l1 l2 l3, alt l1 l2 l3 → alternate l1 l2 = l3.
Proof.
induction l1; intros.
- inversion H.
  subst.
  simpl.
  reflexivity.
- destruct l2; simpl.
  * inversion H.
    inversion H4.
    auto.
  * inversion H.
    inversion H4.
    apply IHl1 in H9.
    rewrite H9.
    reflexivity.
Qed.
```

Proving Properties Compactly Using Induction

```
Lemma alternate_alt :
 forall l1 l2 l3, alternate l1 l2 = l3 → alt l1 l2 l3.
Proof.
induction l1; simpl; intros.
- rewrite H. apply alt_nil.
- destruct l2; subst; apply alt_step; try apply alt_nil.
 apply alt_step. apply IHl1. reflexivity.
Qed.
```

Extracting a Verified Function to OCaml

```
Extraction Language Ocaml.

Require Import ExtrOcamlBasic.
Require Import ExtrOcamlNatInt.

Extraction "alternate.ml" alternate.
```

OCaml Result

```
(** val alternate : int list -> int list -> int list **)

let rec alternate l1 l2 =
  match l1 with
  | [] -> l2
  | h1 :: t1 ->
    (match l2 with
     | [] -> h1 :: t1
     | h2 :: t2 -> h1 :: (h2 :: (alternate t1 t2)))
```

Simplified Function?

```
Fixpoint alternate' (l1 l2 : list nat) : list nat :=
match l1 with
| [] ⇒ l2
| h1 :: t1 ⇒ h1 :: alternate' l2 t1
end.
```

One Encoding Option

```
Program Fixpoint alternate' (l1 l2 : list nat)
 { measure (length (l1 ++ l2)) } : { l | alt l1 l2 l } :=
match l1 with
| [] ⇒ l2
| h1 :: t1 ⇒ h1 :: alternate' l2 t1
end.
```