Introduction to Coq

Proving with computer assistance

Adam Koprowski

February 2007

http://www.win.tue.nl/~akoprows/teaching/Coq
A.Koprowski@tue.nl
HG 6.78

technische universiteit eindhoven

Schedule

- Introduction to Coq. (Wed 14/02, 10:45-12:30)
- Lab sessions – exercises with Coq (**please subscribe!**).
  - Wed 28/02, 10:45-12:30 (3+4), MATRIX 1.41 **(full)**
  - Wed 28/02, 15:30-17:15 (7+8), HG 5.95
  - Thu 1/03, 13:30-15:15 (5+6), HG 5.95
- Summary of Coq + short introduction to PVS. (Wed 7/03, 10:45-12:30)

Computer assistance in proofs

Proof consists of:

- *reasoning* (derivation rules) and
- *computations* (reductions).

Computers can assist with both:

- Computations:
  - numerical computations ("calculators")
  - symbolic computations (computer algebra systems)
- Reasoning:

Automated theorem provers	Proof assistants
fully automated system delivers a proof specialized	interactive human delivers a proof highly general

- Under development: combining computations and reasoning in one system (e.g. Maple mode for Coq)

Automated theorem provers vs Proof assistants

Full automation: given a proposition, the program crunches for a bit and then replies "TRUE" (and here's a proof) or "FALSE".

This presupposes an algorithm that for each proposition can determine whether it is provable or not...
in other words: the underlying logic must be decidable.
Ex.: propositional logic, decidable parts of first order logics.
($\Rightarrow$ techniques are topic of 2R880: Automated Reasoning)

However, decidable logics are not very expressive.
More expressive logics are undecidable.
$\Rightarrow$ general proof assistants are necessarily interactive.

Some existing tools:

- Computer Algebra systems: (symbolic) computations
  - Maple, Mathematica, Derive, ...
- Automated Theorem Provers (ATPs):
  - many, mostly very domain-specific
  - ACL2, Otter, Vampire, Waldmeister, ...
- Proof Assistants (PAs):
  - PVS, Coq, HOL (light), Isabelle, Mizar, Agda, LEGO, ...

What are proof assistants good for?

Proof Assistants (PA's) assist in several ways:

- With formalization of theories (giving definitions, axioms etc.)
    - in a suitable general language (e.g. λC, set theory)
    - which is parsed and type checked by the PA.
- With checking and creating proofs
    - in a language for proofs (e.g. λC, natural deduction)
    - by automation of proof-strategies (tactics)
    - interactively with the user
- By providing (administrative) tools
    - editor, library, documentation, search tools
    - program extraction from constructive proofs

First proof assistant: AUTOMATH (esp. formalization & checking proofs)

Applications of Proof Assistants

- Formal(izing) mathematics:
  - QED project; Idea: computer-based encyclopedia and database of all mathematical knowledge.
  - Mizar: *Journal of Formalized Mathematics*
    189 authors, >40.000 theorems.
  - Coq: C-CorRN (Constructive Coq Repository at Nijmegen), e.g. Fundamental Theorem of Algebra
- Verification, e.g.: Testing ············ Modeling ············ Theorem proving

  ⟵Effort—— ——Reliability⟶

  - Software: e.g. LOOPtool for Java and JavaCard (PVS), Krakatoa for Java/JML (Coq), Jive (PVS+Isabelle)
  - Hardware: e.g. HOL light at Intel
- Program extraction (Coq, Isabelle)

Principles of (type theory based) PAs

Central principle: Curry-Howard isomorphism
propositions as types, proofs as terms (already in AUTOMATH)

Type checking ($\Gamma \overset{?}{\vdash} M : \sigma$): decidable
Inhabitation ($\Gamma \vdash ? : \sigma$): undecidable

proof	$\Longleftrightarrow$	term
proposition	$\Longleftrightarrow$	type
proof checking	$\Longleftrightarrow$	type checking
proving / proof search	$\Longleftrightarrow$	term search
program, algorithm	$\Longleftrightarrow$	term
specification	$\Longleftrightarrow$	type
program	$\Longleftrightarrow$	proof

Reliability of proof assistants.

Why should we trust proof assistants?
De Bruijn criterion for reliability (for all proof assistants):

> *Proof(term)s may be created by programs of arbitrary complexity, but there should be a very small and manually verifiable part of the program (kernel) to check those proof(term)s.*

One still may ask:

- What if the hardware is flawed?
  $\Rightarrow$ test on many different architectures
- What if the compiler used to build PA is flawed?
  $\Rightarrow$ compilers are about the most thoroughly tested pieces of software we have
- What if what you are formalizing is different that what you have in mind (and want to prove)?
  $\Rightarrow$ definitions are much easier (and shorter) than proofs; some experience required

So we will never have absolute certainty but it seems that

PAs are as close as we can get

1. Introduction
   - What is a proof assistant?
   - Full automation versus interaction
   - Applications of PAs

2. The Coq proof assistant
   - Introduction
   - Demo

Coq: short intro

- Home: http://coq.inria.fr
  LogiCal project, INRIA, Paris, France
- Based on the Calculus of Inductive Constructions (CIC): λC plus 'inductive types'
- Available for all major platforms (Linux, MS Win, OSX)
- Written in the language OCaml (also an INRIA-product)
- Version 8.0 brought in significant improvements: more powerful, better syntax and more user-friendly (CoqIDE)
- Current version 8.1 (Feb 2007)
- Special feature: program extraction from constructive proofs

Working with Coq: interface

Working directly with Coq's command line interface is next to impossible.
One needs an interface for that and there are two major options:

- CoqIDE (user-friendly but **not stable** for Windows)
- ProofGeneral (uses XEmacs, somehow more difficult to use but stable)

Notations in Coq

λC	Coq
$\star_p$	`Prop`
$\star_s$	`Set`
$\Box$	`Type`
$\lambda x : A.M$	`fun x:A => M`
$\Pi x : A.M$	`forall x:A, M`
$\rightarrow$	`->`

Specification language: Gallina.
Commands are capitalized, tactics aren't.
Lines always end with full-stop (=".").

1. Introduction
   - What is a proof assistant?
   - Full automation versus interaction
   - Applications of PAs
2. The Coq proof assistant
   - Introduction
   - Demo

Check/Print

The command `Check` produces the type of its argument, `Print` produces information on how its argument is defined.

```
> Check nat.
nat:  Set.
> Check fun n:nat => (plus n (S 0))
fun n:nat => n + 1 :  nat -> nat

> Print nat.
Inductive nat:  Set := 0:nat | S:nat->nat
```

About the libraries

When you start Coq the *Core library* is loaded at start.
It defines many basic notions and notations (e.g. `Set`, `nat`, `plus`, `+`, `lt`, `<`).
For more involved properties and definitions one may need to load other libraries. E.g.:

```
> Require Import Arith.
```

You may then find properties using the commands `SearchAbout`, or `SearchPattern`, e.g.:

```
> SearchAbout lt.
```

Declarations/Definitions

Variables can be declared as follows:

```
> Variable n: nat.
```

One may assume properties for declared variables:

```
> Hypothesis Pos_n: n>0.
```

Definitions look like this:

```
> Definition double := fun x: nat => (plus x
x).
> Definition double' (x: nat) := x + x.
```

Proofs

`> Goal forall A B: Prop, A->B->A.` ← starts proof mode

[proof mode: apply tactics until `Proof completed.`]

`> Save ABA.` ← save the proof term as `ABA`.

One can also introduce a goal as Lemma or Theorem:

`> Lemma NAME: forall A B:Prop, A->B->A.`

`> Proof.` ← (not strictly necessary)

[apply tactics until `Proof completed.`]

`> Qed.` ← saves the proof term as `NAME`.

(To postpone a certain proof obligation, one can use `Admitted`.)

intro

```
                                ...
...              →              H:A
-----       (intro H.)          -------
A->B                            B
```

```
                                                  x:U
                                   →              H:A
--------------------          (intros.)           -------
forall x:U, A->B                                  B
```

assumption

If `A` and `B` are convertible, then:

```
...
H:B
...
―――――――        →        Subgoal completed
A        (assumption.)
```

unfold

`Definition two:= (S (S 0)).`

`...`
 ———————
 `two=(S (S 0))`
	$\longrightarrow$
	 `(unfold two.)`
		`...`
		 ———————
		 `(S (S 0))=(S (S 0))`

See also `fold`.

apply

```
H: forall x:U,                      H: forall x:U,
   B(x)->C(x)         →                B(x)->C(x)
...             (apply H.)          ...
_______________                     _______________
C(t)                                B(t)
```

elim

H: A $\wedge$ B ... ——————— C	(elim H.) $\longrightarrow$	H: A $\wedge$ B ... ——————— A->B->C

Recall that A $\wedge$ B is defined as follows in λC:

```
forall C, (A->B->C)->C
```

rewrite

`H: a=b`
`...`
———
`(P b)`
	`(Rewrite <- H.)`
	$\longrightarrow$
		`H: a=b`
		`...`
		———
		`(P a)`

simpl

```
...                                         ...
...                                         ...
________________                            ________________
(P ((fun x:A=>x)b))          →              (P b)
                         (simpl.)
```

β-reduction!

induction

$\dfrac{\ldots}{\texttt{forall n:nat, (P n)}}$	$\xrightarrow{\texttt{(induction n.)}}$	$\dfrac{\ldots}{\texttt{(P 0)}}$ $\dfrac{\ldots}{\texttt{forall n:nat, (P n)->(P (S n))}}$

Works for inductively defined types. For example:

```
Inductive nat:Set := 0:nat | S: nat->nat.
```

Some other useful basic tactics

- `split`: to split a conjunctive goal into two subgoals
- `left`/`right`: to prove one side of a disjunctive goals
- `reflexivity`: to prove a goal of the form `x = x`.
- `auto`, `trivial`, `intuition`
- `case n`: for a case distinction on a variable
  (Demo: definitions with case distinction?)

Useful reference (table on p. 7!): “Coq in a Hurry”
by Yves Bertot (see course-webpage for link)

"Homework"

In two weeks we have lab session (28/02 or 1/03).
Please bring your laptop with:

- full battery,
- pre-installed Coq and CoqIDE or Proof General interface (**instructions on course web-page**),
- the file CoqLab.v (**from course web-page**) and
- strongly recommended: read "Coq in a Hurry" (**again, available on course web-page)**