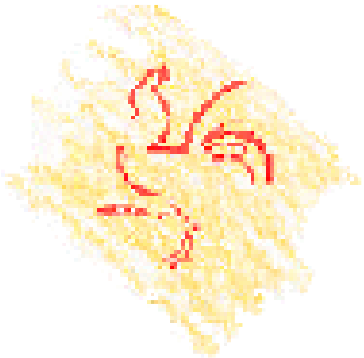
NICTA

COMP 4161

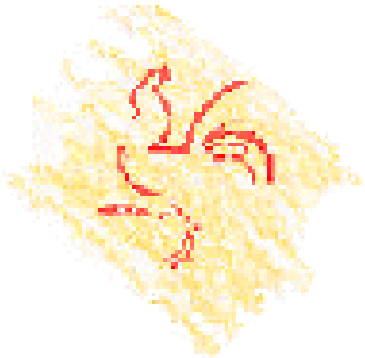
NICTA Advanced Course

Advanced Topics in Software Verification



June Andronick

Formal Methods



Coq

Content

- Introduction
- Curry-Howard Isomorphism
- Coq Language
- Deduction rules
- Proofs in Coq
- Inductive definitions

INTRODUCTION

Basics

Main componants:

- Logical Language

  Calculus of Inductive Constructions (CIC)

- Proof Assistant

  Incremental construction of proof trees

- Program Extractor

  Possibility to extract programs from the constructive contents of proofs.

Exploiting the notion of Curry-Howard isomorphim

Credits

1985 CoC - Calculus of Constructions, T. Coquand

→ Most expressive λ-calculus
(polymorphism, dependant types and higher order)

→ Exploiting Curry-Howard isomorphim

→ Limitation: no direct inductive definitions (functional encodings needed)

1989 CIC - Calculus of Inductive Constructions, T. Coquand, C. Paulin

⟶ Extension with primitive inductive definitions

1989 First Release of Coq (CoC), G. Huet, T. Coquand

1989-2008 G. Dowek, C. Paulin, B. Werner, H. Herbelin, B. Barras, J-C. Fillitre, J. Courant, D. Delahaye, P. Loiseleur. . .








. . . . . .

CURRY-HOWARD ISOMORPHISM

Curry-Howard Isomorphism - Intuition

Intuition: Similarities between typing and logic

$$\frac{}{\Gamma \vdash x : \tau}\ \text{var} \quad \text{if } [x : \tau] \in \Gamma$$

$$\frac{}{\Gamma \vdash A}\ \text{ax} \quad \text{if } A \in \Gamma$$

$$\frac{\Gamma \vdash t_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1\ t_2) : \tau_1}\ \text{app}$$

$$\frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \text{impD}$$

$$\frac{\Gamma \cup [x : \tau_1] \vdash t : \tau_2}{\Gamma \vdash (\lambda x.\ t) : \tau_1 \to \tau_2}\ \text{abs}$$

$$\frac{\Gamma \cup [A] \vdash B}{\Gamma \vdash A \to B}\ \text{impI}$$

Curry-Howard Isomorphism - Intuition

Intuition: Similarities between typing and logic

$$\frac{}{\Gamma \vdash x : \tau}\ \text{var} \quad \text{if } [x : \tau] \in \Gamma \qquad \frac{}{\Gamma \vdash A}\ \text{ax} \quad \text{if } A \in \Gamma$$

$$\frac{\Gamma \vdash t_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1\ t_2) : \tau_1}\ \text{app} \qquad \frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \text{impD}$$

$$\frac{\Gamma \cup [x : \tau_1] \vdash t : \tau_2}{\Gamma \vdash (\lambda x.\ t) : \tau_1 \to \tau_2}\ \text{abs} \qquad \frac{\Gamma \cup [A] \vdash B}{\Gamma \vdash A \to B}\ \text{impI}$$

type $\longleftrightarrow$ proposition

term $\longleftrightarrow$ **proof**

Curry-Howard Isomorphism - The Core

NICTA

type $\longleftrightarrow$ proposition

term $\longleftrightarrow$ proof

$T : \tau$ is a term T of type τ

a proof T of the proposition τ

Proving a proposition (i.e. a type) is building an inhabitant of this type

Here comes the notion of *proof term*.

Playing with Curry-Howard Isomorphism

NICTA

$A \to B$ is the type of a function that associates

a term of type B to any term of type A

a proof of the proposition A to any proof of the proposition B

A proof of $A \to B$ is a term p of type $A \to B$

a term of the form $(\lambda x.\ t)$

where x is a proof of A and t a proof of B

Derivation rules merge with typing rules:

$$\frac{\Gamma \cup [x : A] \vdash t : B}{\Gamma \vdash (\lambda x.\ t) : A \to B}$$

Checking a proof is type-checking the proof term

COQ LANGUAGE

COQ LANGUAGE

Idea: everything is a term ! . . . even types !

This gives

- polymophism

  Examples: $eq : \ \forall \tau.\ \tau \rightarrow \tau \rightarrow Prop$ (Isabelle: **eq::'a→'a→ bool**)

  $refl\_eq : \forall \tau.\ \forall x : \tau.\ x = x$ (Isabelle's refl:**?x=?x**)

- dependant types

  Examples: $\forall n : nat.\ \ (list\ n)$

  $\forall n : nat.\ \ n > 1 \ \ \rightarrow \ \ (list\ n)$

  $append : \ \ \forall n : nat.\ \ (list\ n) \rightarrow$

  $\forall m : nat.\ \ (list\ m) \rightarrow (list\ (n + m))$

(no type variable, no schematic, no free variable, only terms and forall)

COQ LANGUAGE

Idea: everything is a term ! . . . even types !

Problem: if a type is a term, then it has a type !

Indeed.

- The type of a type is always a constant of the language called a sort.
- The two basic sorts in the language are:
  - **Prop**: type of propositions
  - **Set**: type of specifications (programs + usual sets - bool, nat, etc)

OK but **Set** and **Prop** are types. . . so they are terms. . . so they have a type!

Prop: Type **Set: Type**

OK but **Type** is a type. . . . . .

Type=Type(0) **Type(i) :Type (i+1)**

. . . don't ask me more !

NICTA

Terms: $t ::= \quad v \mid c \mid (t\ t) \mid (\lambda x : t.\ t) \mid$

Prop | **Set** | **Type** |

$(\forall x : t.\ t)$

$(\forall x : A.\ B)$ is the dependent product

It may represent any of the following:

$(\forall n : nat.\ n + 0 = n)$

$(\forall n : nat.\ (list\ n))$

$(\forall A : \textbf{Prop}.\ A = A \vee A)$

$nat \rightarrow bool$ $\qquad$ $(\forall n : nat.\ bool)$

$A \rightarrow (A \rightarrow B) \rightarrow B$ $\qquad$ $(\forall pa : A.(\forall\ pab : A \rightarrow B.\ B))$

$(\forall x : A.\ B)$ is written $A \rightarrow B$ if B does not depend on x

Deduction rules

Deduction rules

Only one intro rule, one apply rule and one assumption rule:

$$\frac{}{\Gamma \vdash x : A}$$ assumption if $[x : A] \in \Gamma$

$$\frac{\Gamma \cup [x : T] \vdash t : U}{\Gamma \vdash (\lambda x : T.\ t) : \forall x : T.\ U}$$ intro (dependent)

$$\frac{\Gamma \vdash t : (\forall x : U.\ T) \quad \Gamma \vdash u : U}{\Gamma \vdash (t\ u) : T[x \leftarrow u]}$$ apply (dependent)

$$\frac{\Gamma \cup [x : T] \vdash t : U}{\Gamma \vdash (\lambda x : T.\ t) : T \to U}$$ intro (non dependent)

$$\frac{\Gamma \vdash t : U \to T \quad \Gamma \vdash u : U}{\Gamma \vdash (t\ u) : T}$$ apply (non dependent)

Proofs in Coq

Vernacular Commands

Display and Request the Environment

`Check`	gives	the type of a term	`( Check true. )`
		the statement of a proof name	`( Check refl_equal. )`
		the type of a type	`( Check bool. )`
`Print`	gives	the definition of a variable	`( Print true. )`
		the proof term of a proof name	`( Print refl_equal. )`
		the definition of a type	`( Print bool. )`

`Search` `( Search eq. )`

`Locate` `( Locate nat. )`

DEMO

Proofs in Coq

General Schema:

Lemma name: <proposition>.
Proof.
<tactic>.
<tactic>.
...
Qed.

Alternatives: `Lemma/Theorem/Remark/Fact` or `Goal` (no name)
`Defined/Save/Admitted`

Notations:

$\lambda x : T.\ t$	is written	`fun x:T ⇒ t`
$\forall x : T.\ t$	is written	`forall x:T, t`
$\forall x : T.\ \forall y : T.\ t$	is written	`forall x y :T, t`
$\forall x : T.\ \forall y : U.\ t$	is written	`forall (x:T)(y:U), t`

Tactics `assumption`, `intro` and `apply`:

$$\frac{}{\Gamma \vdash \mathtt{x} : \mathtt{A}}\ \texttt{assumption if } [\mathtt{x} : \mathtt{A}] \in \Gamma$$

$$\frac{\Gamma \cup [\mathtt{x} : \mathtt{T}] \vdash \mathtt{t} : \mathtt{U}}{\Gamma \vdash (\mathtt{fun}\ \mathtt{x} : \mathtt{T} \Rightarrow \mathtt{t}) : \mathtt{T} \to \mathtt{U}}\ \texttt{intro}$$

$$\frac{\Gamma \cup [\mathtt{x} : \mathtt{T}] \vdash \mathtt{t} : \mathtt{U}}{\Gamma \vdash (\mathtt{fun}\ \mathtt{x} : \mathtt{T} \Rightarrow \mathtt{t}) : \mathtt{forall}\ \mathtt{x} : \mathtt{T},\ \mathtt{U}}\ \texttt{intro}$$

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{U} \to \mathtt{T} \quad \Gamma \vdash \mathtt{u} : \mathtt{U}}{\Gamma \vdash (\mathtt{t}\ \mathtt{u}) : \mathtt{T}}\ \texttt{apply t}$$

DEMO

TACTICAL

`tactic`$_1$ `; tactic`$_2$`.`

applies `tactic`$_2$ to all subgoals generated by `tactic`$_1$
(if only one, just chaining)

`tactic`$_0$ `; [tactic`$_1$`|...|tactic`$_n$`].`

applies `tactic`$_i$ to the i-th subgoal generated by `tactic`$_0$

DEMO

INDUCTIVE DEFINITIONS

Inductive (CIC)

Everything is defined inductively.

- Inductive types

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

- Inductively defined functions:

```
Fixpoint plus (n m:nat) {struct n} : nat :=
  match n with
  | O => m
  | S p => S (plus p m)
  end.
```

INDUCTIVE (CIC)

Everything is defined inductively.

- Inductive predicates

```
Inductive is_even : nat-> Prop :=
 Zero_is_even : is_even O
|Succ_even    : forall n:nat,
                 is_even n -> is_even (S(S n)).
```

- proof by induction:

```
induction n.
destruct n.
inversion H.
discriminate.
injection H.
...
```

INDUCTIVE (CIC)

NICTA

But also:

```
Inductive True : Prop :=
  I : True.

Inductive False : Prop :=.

Inductive and (A B:Prop) : Prop :=
  conj : A -> B -> A and B.

Inductive or (A B:Prop) : Prop :=
  | or_introl : A -> A or B
  | or_intror : B -> A or B

...
...
...
```

DEMO

LEARN MORE

http://coq.inria.fr/