Coq - introduction

Daria Walukiewicz-Chrząszcz

07.03.2017

Coq — interactive proof assistant

http://coq.inria.fr/

```
(***********)
(*  v      *)
(* <O___,, *)
(*   \VV/  *)
(*    //   *)
(*         *)
(***********)
```

- rich (pure) functional programming language
- rich logical language
- user writes proofs
- Coq makes sure every step is correct
- and solves subgoals for which automated proving algorithms have been implemented
- (proved to be correct) program can be extracted to Ocaml, Haskell, Scheme...

Coq — interactive proof assistant

http://coq.inria.fr/

```
(***********)
(*  v      *)
(* <O___,, *)
(*   \VV/  *)
(*    //   *)
(*         *)
(***********)
```

- rich (pure) functional programming language
- rich logical language
- user writes proofs
- Coq makes sure every step is correct
- and solves subgoals for which automated proving algorithms have been implemented
- (proved to be correct) program can be extracted to Ocaml, Haskell, Scheme...

Coq — interactive proof assistant

http://coq.inria.fr/

```
(***********)
(*  v      *)
(* <O___,, *)
(*   \VV/  *)
(*    //   *)
(*         *)
(***********)
```

- rich (pure) functional programming language
- rich logical language
- user writes proofs
- Coq makes sure every step is correct
- and solves subgoals for which automated proving algorithms have been implemented
- (proved to be correct) program can be extracted to Ocaml, Haskell, Scheme...

Coq — interactive proof assistant

http://coq.inria.fr/

```
(***********)
(*  v      *)
(* <O___,, *)
(*   \VV/  *)
(*    //   *)
(*         *)
(***********)
```

- rich (pure) functional programming language
- rich logical language
- user writes proofs
- Coq makes sure every step is correct
- and solves subgoals for which automated proving algorithms have been implemented
- (proved to be correct) program can be extracted to Ocaml, Haskell, Scheme...

Coq — interactive proof assistant

http://coq.inria.fr/

```
(***********)
(*  v      *)
(* <O___,, *)
(*   \VV/  *)
(*    //   *)
(*         *)
(***********)
```

- rich (pure) functional programming language
- rich logical language
- user writes proofs
- Coq makes sure every step is correct
- and solves subgoals for which automated proving algorithms have been implemented
- (proved to be correct) program can be extracted to Ocaml, Haskell, Scheme...

Coq — interactive proof assistant

http://coq.inria.fr/

```
(***********)
(*  v      *)
(* <O___,, *)
(*   \VV/  *)
(*    //   *)
(*         *)
(***********)
```

- rich (pure) functional programming language
- rich logical language
- user writes proofs
- Coq makes sure every step is correct
- and solves subgoals for which automated proving algorithms have been implemented
- (proved to be correct) program can be extracted to Ocaml, Haskell, Scheme...

Coq — interactive proof assistant

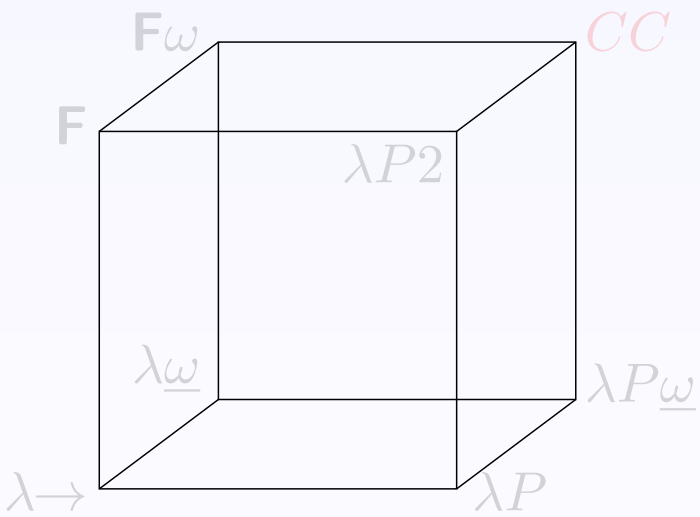
http://coq.inria.fr/

```
(***********)
(*  v      *)
(* <O___,, *)
(*   \VV/  *)
(*    //   *)
(*         *)
(***********)
```

- rich (pure) functional programming language
- rich logical language
- user writes proofs
- Coq makes sure every step is correct
- and solves subgoals for which automated proving algorithms have been implemented
- (proved to be correct) program can be extracted to Ocaml, Haskell, Scheme...

Coq — formalism

Coq — higher order logic (calculus of constructions (CC) + inductive definitions)



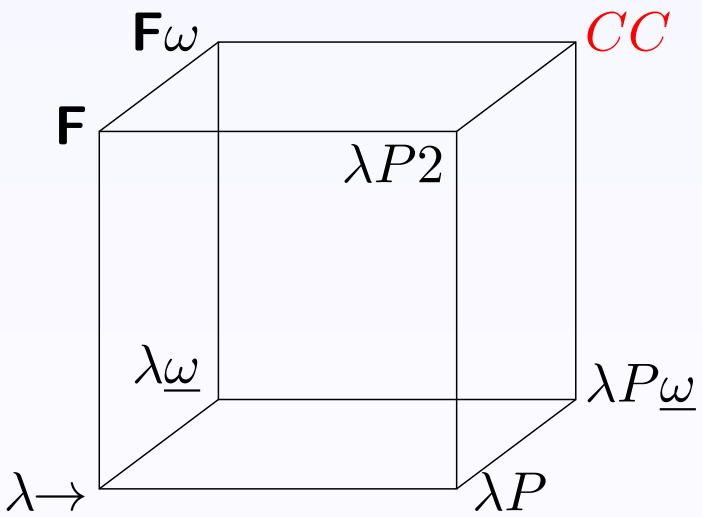
↑ polimorphism

↗ type constructors

→ dependent types

Coq — formalism

Coq — higher order logic (calculus of constructions (CC) + inductive definitions)



↑ polimorphism

↗ type constructors

→ dependent types

Proving in Coq

Curry-Howard isomorphism

proof of a given formula
↕ ↕
term of the corresponding type

$\lambda x^{A \to B \to C} \lambda y^{A \to B} \lambda z^{A}\, xz(yz) \;:\; (A \to B \to C) \to (A \to B) \to (A \to C)$

Correctness of Coq relies on correctness of type-checking

Proving in Coq

Curry-Howard isomorphism

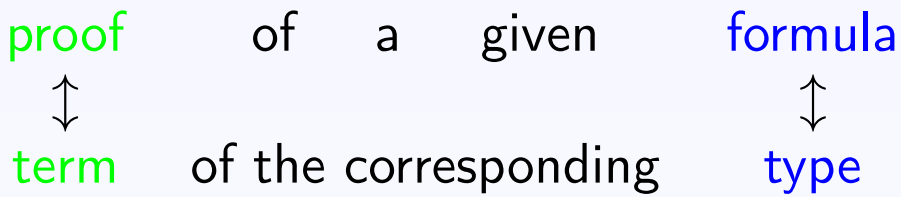
proof	of	a	given	formula
↕				↕
term	of the corresponding			type

$$\lambda x^{A \to B \to C} \lambda y^{A \to B} \lambda z^{A}\, xz(yz) \;:\; (A \to B \to C) \to (A \to B) \to (A \to C)$$

Correctness of Coq relies on correctness of type-checking

Proving in Coq

Curry-Howard isomorphism



$$\lambda x^{A \to B \to C} \lambda y^{A \to B} \lambda z^{A}\ xz(yz) \ : \ (A \to B \to C) \to (A \to B) \to (A \to C)$$

Correctness of Coq relies on correctness of type-checking

Proving in Coq

Curry-Howard isomorphism

proof	of	a	given	formula
↕				↕
term	of the corresponding			type

$$\lambda x^{A \to B \to C} \lambda y^{A \to B} \lambda z^{A}\ xz(yz) \ : \ (A \to B \to C) \to (A \to B) \to (A \to C)$$

Correctness of Coq relies on correctness of type-checking

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Coq — a bit of history

1984 CoC - calculus of constructions - G. Huet, T. Coquand

1989 first public release (version 4.10)

1991 Coq - calculus of inductive constructions - C. Paulin (version 5.6)

⋮

2000 version 7.0 with new (safer) architecture

2003 version 7.4 with modules

2004 version 8.0 with new syntax

2009 version 8.2 with "type classes"

2012 version 8.4 with eta-reduction, structural proof syntax...

2016 version 8.5 and 8.6

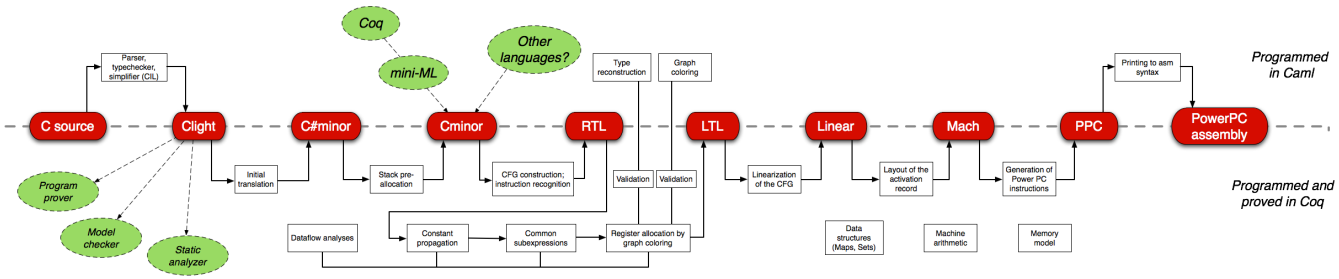
Coq — famous formalizations

- Fundamental theorem of algebra, Nijmegen 2000
- JavaCard Platform formalization, Trusted Logic 2003

  September 2007: a big step in program certification in the real world: The Technology and Innovation group at Gemalto has successfully completed a Common Criteria (CC) evaluation on a JavaCard based commercial product. This evaluation is the world's first CC certificate of a Java product involving EAL7 components. (the official press release)

- Four color theorem, Cambridge 2004
- CompCert certified Clight compiler, 2008-now

  The main result of the project is the CompCert C verified compiler, a high-assurance compiler for almost all of the ISO C90 / ANSI C language, generating efficient code for the PowerPC, ARM and x86 processors.



Coq — programming language

- predicative sorts `Set` and `Type`
- abstraction and application
- inductive types,
- (structural) recursion
- polimorphism
- dependant types and dependent pattern-matching
- modules i functors
- type classes
- ...

Coq — programming language

- predicative sorts `Set` and `Type`
- abstraction and application
- inductive types,
- (structural) recursion
- polimorphism
- dependant types and dependent pattern-matching
- modules i functors
- type classes
- ...

Coq — programming language

- predicative sorts `Set` and `Type`
- abstraction and application
- inductive types,
- (structural) recursion
- polimorphism
- dependant types and dependent pattern-matching
- modules i functors
- type classes
- ...

Coq — programming language

- predicative sorts `Set` and `Type`
- abstraction and application
- inductive types,
- (structural) recursion
- polimorphism
- dependant types and dependent pattern-matching
- modules i functors
- type classes
- ...

Coq — programming language

- predicative sorts `Set` and `Type`
- abstraction and application
- inductive types,
- (structural) recursion
- polimorphism
- dependant types and dependent pattern-matching
- modules i functors
- type classes
- ...

Coq — programming language

- predicative sorts `Set` and `Type`
- abstraction and application
- inductive types,
- (structural) recursion
- polimorphism
- dependant types and dependent pattern-matching
- modules i functors
- type classes
- ...

Coq — programming language

- predicative sorts `Set` and `Type`
- abstraction and application
- inductive types,
- (structural) recursion
- polimorphism
- dependant types and dependent pattern-matching
- modules i functors
- type classes
- ...

Coq — programming language

- predicative sorts `Set` and `Type`
- abstraction and application
- inductive types,
- (structural) recursion
- polimorphism
- dependant types and dependent pattern-matching
- modules i functors
- type classes
- ...

Coq — programming language

- predicative sorts `Set` and `Type`
- abstraction and application
- inductive types,
- (structural) recursion
- polimorphism
- dependant types and dependent pattern-matching
- modules i functors
- type classes
- ...

Coq — logic

- intuitionistic higher-order logic
- impredicative sort `Prop`
- forall and implication built-in
- boolean connectives, false, exists (defined)
- inductive predicates (including equality)
- ...

Coq — logic

- intuitionistic higher-order logic
- impredicative sort `Prop`
- forall and implication built-in
- boolean connectives, false, exists (defined)
- inductive predicates (including equality)
- ...

Coq — logic

- intuitionistic higher-order logic
- impredicative sort `Prop`
- forall and implication built-in
- boolean connectives, false, exists (defined)
- inductive predicates (including equality)
- ...

Coq — logic

- intuitionistic higher-order logic
- impredicative sort `Prop`
- forall and implication built-in
- boolean connectives, false, exists (defined)
- inductive predicates (including equality)
- ...

Coq — logic

- intuitionistic higher-order logic
- impredicative sort `Prop`
- forall and implication built-in
- boolean connectives, false, exists (defined)
- inductive predicates (including equality)
- ...

Coq — logic

- intuitionistic higher-order logic
- impredicative sort `Prop`
- forall and implication built-in
- boolean connectives, false, exists (defined)
- inductive predicates (including equality)
- ...

Coq proof machinery

- interactive proof mode (goal management)
- built-in tactics (constructing a bit of proof-term): intro, apply, etc.
- automatic ad-hoc tactics: auto, intuition, etc.
- decision procedures: omega, ring, field, tauto, etc.
- tactic language (`Ltac mytactic:=...`)

Coq proof machinery

- interactive proof mode (goal management)
- built-in tactics (constructing a bit of proof-term): intro, apply, etc.
- automatic ad-hoc tactics: auto, intuition, etc.
- decision procedures: omega, ring, field, tauto, etc.
- tactic language (`Ltac mytactic:=...`)

Coq proof machinery

- interactive proof mode (goal management)
- built-in tactics (constructing a bit of proof-term): intro, apply, etc.
- automatic ad-hoc tactics: auto, intuition, etc.
- decision procedures: omega, ring, field, tauto, etc.
- tactic language (`Ltac mytactic:=...`)

Coq proof machinery

- interactive proof mode (goal management)
- built-in tactics (constructing a bit of proof-term): intro, apply, etc.
- automatic ad-hoc tactics: auto, intuition, etc.
- decision procedures: omega, ring, field, tauto, etc.
- tactic language (`Ltac mytactic:=...`)

Coq proof machinery

- interactive proof mode (goal management)
- built-in tactics (constructing a bit of proof-term): intro, apply, etc.
- automatic ad-hoc tactics: auto, intuition, etc.
- decision procedures: omega, ring, field, tauto, etc.
- tactic language (`Ltac mytactic:=...`)

Program extraction

- program is extracted from the proof
- extracted program satisfies its specification *by definition*
- extraction — "elimination" of logical parts from the proof-term
- extraction possible because proofs are done in constructive logic (excluded-middle and double negation laws do not hold)
- target languages: O'Caml, Haskell, Scheme

Program extraction

- program is extracted from the proof
- extracted program satisfies its specification *by definition*
- extraction — "elimination" of logical parts from the proof-term
- extraction possible because proofs are done in constructive logic (excluded-middle and double negation laws do not hold)
- target languages: O'Caml, Haskell, Scheme

Program extraction

- program is extracted from the proof
- extracted program satisfies its specification *by definition*
- extraction — "elimination" of logical parts from the proof-term
- extraction possible because proofs are done in constructive logic (excluded-middle and double negation laws do not hold)
- target languages: O'Caml, Haskell, Scheme

Program extraction

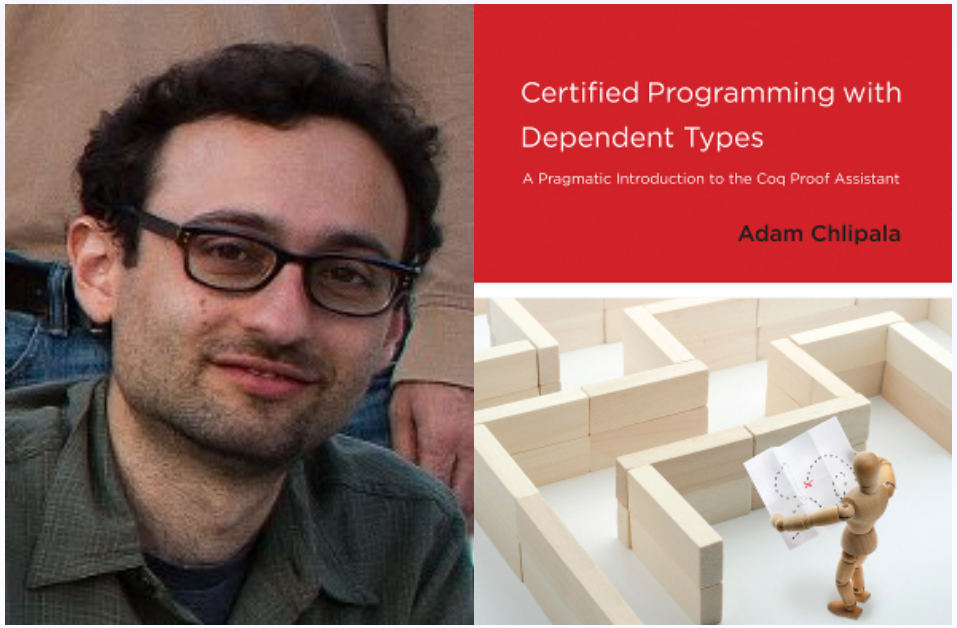
- program is extracted from the proof
- extracted program satisfies its specification *by definition*
- extraction — "elimination" of logical parts from the proof-term
- extraction possible because proofs are done in constructive logic (excluded-middle and double negation laws do not hold)
- target languages: O'Caml, Haskell, Scheme

Program extraction

- program is extracted from the proof
- extracted program satisfies its specification *by definition*
- extraction — "elimination" of logical parts from the proof-term
- extraction possible because proofs are done in constructive logic (excluded-middle and double negation laws do not hold)
- target languages: O'Caml, Haskell, Scheme

Resources

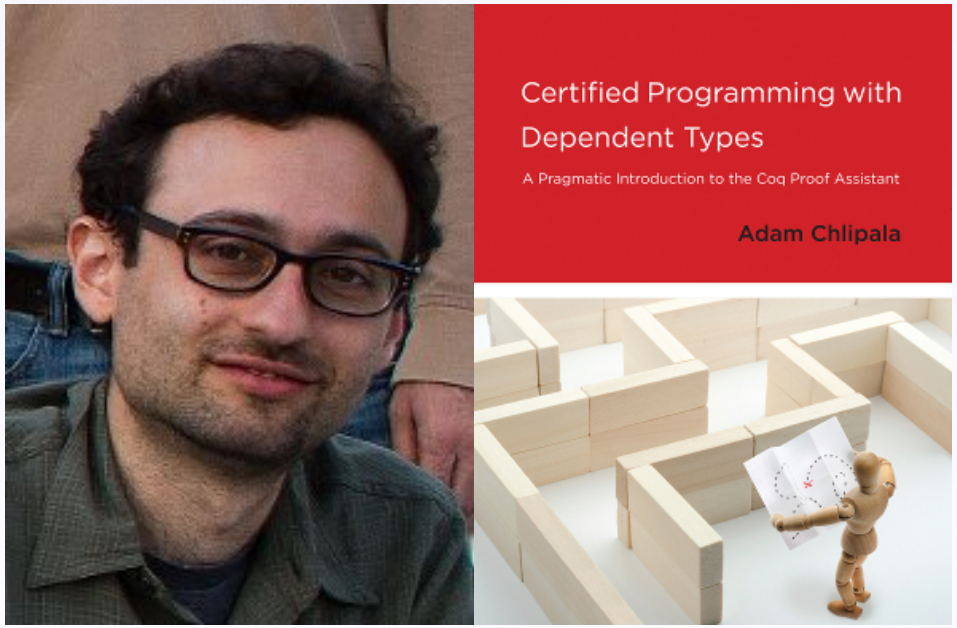
- `https://coq.inria.fr/`
- Coq Art, Yves Bertot, Pierre Castéran
- Certified Programming with Dependent Types, Adam Chlipala (MIT)



editor: MIT Press 2013
accessible: `http://adam.chlipala.net/cpdt/`

Resources

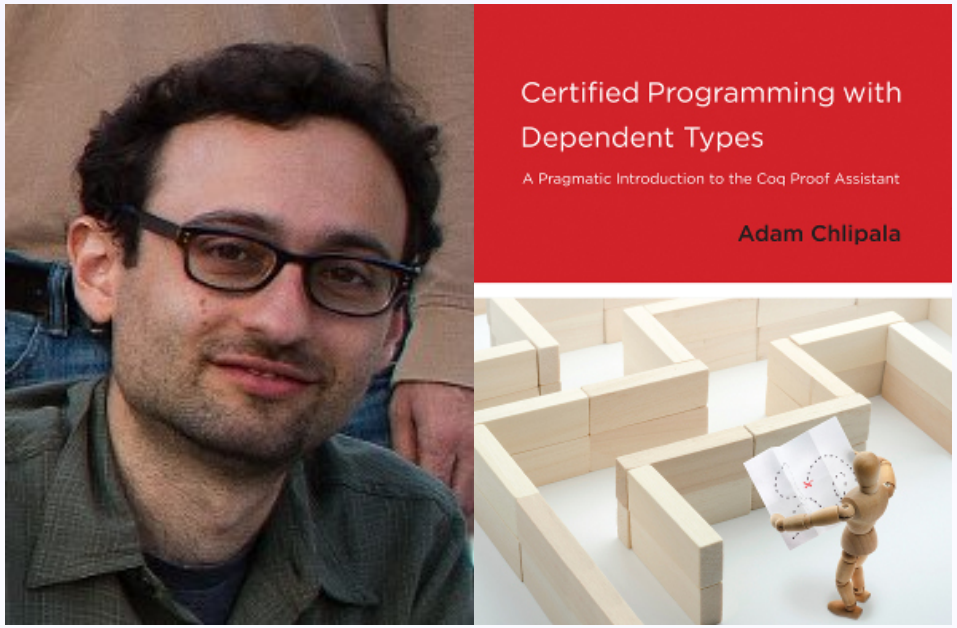
- `https://coq.inria.fr/`
- Coq Art, Yves Bertot, Pierre Castéran
- Certified Programming with Dependent Types, Adam Chlipala (MIT)



editor: MIT Press 2013
accessible: `http://adam.chlipala.net/cpdt/`

Resources

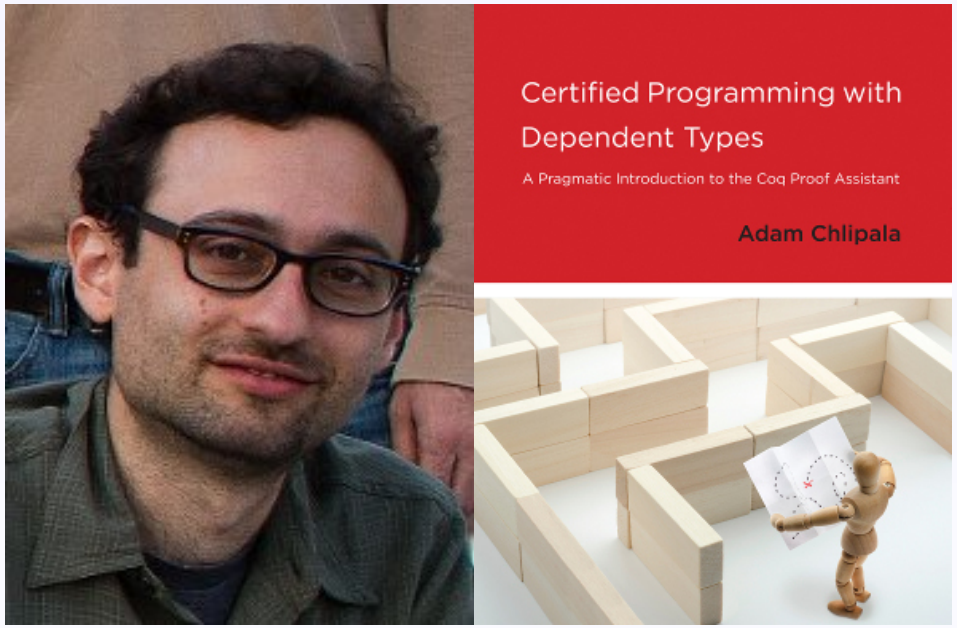
- `https://coq.inria.fr/`
- Coq Art, Yves Bertot, Pierre Castéran
- Certified Programming with Dependent Types, Adam Chlipala (MIT)



editor: MIT Press 2013
accessible: `http://adam.chlipala.net/cpdt/`

Resources

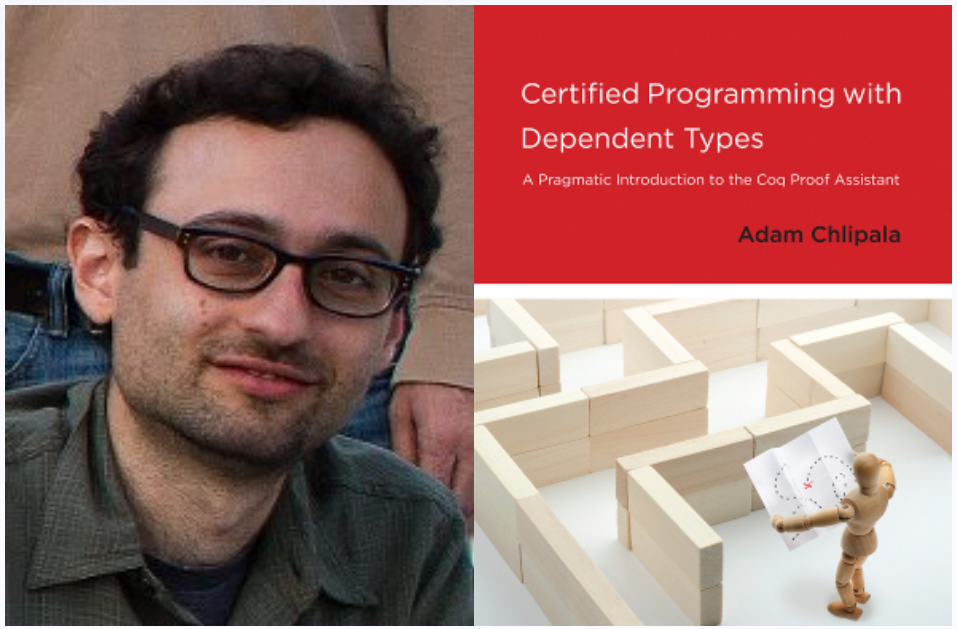
- `https://coq.inria.fr/`
- Coq Art, Yves Bertot, Pierre Castéran
- Certified Programming with Dependent Types, Adam Chlipala (MIT)



editor: MIT Press 2013
accessible: `http://adam.chlipala.net/cpdt/`

Resources

- `https://coq.inria.fr/`
- Coq Art, Yves Bertot, Pierre Castéran
- Certified Programming with Dependent Types, Adam Chlipala (MIT)



editor: MIT Press 2013
accessible: `http://adam.chlipala.net/cpdt/`

Typing

$$\texttt{environment} \vdash \texttt{term} : \texttt{type}$$

environment: global and local declarations and definitions

types are terms and have types, ex. `nat:Set`, `Set:Type`$_1$

but there are terms that are not types, ex. `fun n:nat => n`

Typing

$$\texttt{environment} \vdash \texttt{term} : \texttt{type}$$

environment: global and local declarations and definitions

types are terms and have types, ex. `nat:Set`, `Set:Type`$_1$

but there are terms that are not types, ex. `fun n:nat => n`

Typing

$$\texttt{environment} \vdash \texttt{term} : \texttt{type}$$

environment: global and local declarations and definitions

types are terms and have types, ex. `nat:Set`, `Set:Type`$_1$

but there are terms that are not types, ex. `fun n:nat => n`

Typing

$$\texttt{environment} \vdash \texttt{term} : \texttt{type}$$

environment: global and local declarations and definitions

types are terms and have types, ex. `nat:Set`, `Set:Type`$_1$

but there are terms that are not types, ex. `fun n:nat => n`

Coq — formalism: fun for all

simple types abstraction rule:

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M \ : \ A \to B}$$

dependent types abstraction rule:

$$\frac{\Gamma, x{:}A \vdash M : B(x)}{\Gamma \vdash \lambda x{:}A.M \ : \ \forall x{:}A.B(x)}$$

Shorthand: $A \to B$ is $\forall x{:}A.B$, where $x \notin FV(B)$

concrete Coq syntax:

`fun n:nat =>` M `: forall n:nat, vector n`

application rule:

$$\frac{\Gamma \vdash F : A \to B \quad \Gamma \vdash G : A}{\Gamma \vdash F\,G : B} \qquad \frac{\Gamma \vdash F : \forall x{:}A.B(x) \quad \Gamma \vdash G : A}{\Gamma \vdash F\,G : B[G/x]}$$

Coq — formalism: fun for all

simple types abstraction rule:

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M \ : \ A \to B}$$

dependent types abstraction rule:

$$\frac{\Gamma, x{:}A \vdash M : B(x)}{\Gamma \vdash \lambda x{:}A.M \ : \ \Pi x{:}A.B(x)}$$

Shorthand: $A \to B$ is $\forall x{:}A.B$, where $x \notin FV(B)$

concrete Coq syntax:

`fun n:nat =>` M `: forall n:nat, vector n`

application rule:

$$\frac{\Gamma \vdash F : A \to B \quad \Gamma \vdash G : A}{\Gamma \vdash F\,G : B} \qquad \frac{\Gamma \vdash F : \forall x{:}A.B(x) \quad \Gamma \vdash G : A}{\Gamma \vdash F\,G : B[G/x]}$$

Coq — formalism: fun for all

simple types abstraction rule:

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M \ : \ A \to B}$$

dependent types abstraction rule:

$$\frac{\Gamma, x{:}A \vdash M : B(x)}{\Gamma \vdash \lambda x{:}A.M \ : \ \forall x{:}A.B(x)}$$

Shorthand: $A \to B$ is $\forall x{:}A.B$, where $x \notin FV(B)$

concrete Coq syntax:

`fun n:nat =>` M `: forall n:nat, vector n`

application rule:

$$\frac{\Gamma \vdash F : A \to B \quad \Gamma \vdash G : A}{\Gamma \vdash F\,G : B} \qquad \frac{\Gamma \vdash F : \forall x{:}A.B(x) \quad \Gamma \vdash G : A}{\Gamma \vdash F\,G : B[G/x]}$$

Coq — formalism: fun for all

simple types abstraction rule:

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M \ : \ A \to B}$$

dependent types abstraction rule:

$$\frac{\Gamma, x{:}A \vdash M : B(x)}{\Gamma \vdash \lambda x{:}A.M \ : \ \forall x{:}A.B(x)}$$

Shorthand: $A \to B$ is $\forall x{:}A.B$, where $x \notin FV(B)$

concrete Coq syntax:

`fun n:nat =>` M `: forall n:nat, vector n`

application rule:

$$\frac{\Gamma \vdash F : A \to B \quad \Gamma \vdash G : A}{\Gamma \vdash F\,G : B} \qquad \frac{\Gamma \vdash F : \forall x{:}A.B(x) \quad \Gamma \vdash G : A}{\Gamma \vdash F\,G : B[G/x]}$$

Coq — formalism: fun for all

simple types abstraction rule:

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M \ : \ A \to B}$$

dependent types abstraction rule:

$$\frac{\Gamma, x{:}A \vdash M : B(x)}{\Gamma \vdash \lambda x{:}A.M \ : \ \forall x{:}A.B(x)}$$

Shorthand: $A \to B$ is $\forall x{:}A.B$, where $x \notin FV(B)$

concrete Coq syntax:

`fun n:nat =>` M `:` `forall n:nat, vector n`

application rule:

$$\frac{\Gamma \vdash F : A \to B \quad \Gamma \vdash G : A}{\Gamma \vdash F\,G : B} \qquad \frac{\Gamma \vdash F : \forall x{:}A.B(x) \quad \Gamma \vdash G : A}{\Gamma \vdash F\,G : B[G/x]}$$

Coq — formalism: fun for all

simple types abstraction rule:

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M \ : \ A \to B}$$

dependent types abstraction rule:

$$\frac{\Gamma, x{:}A \vdash M : B(x)}{\Gamma \vdash \lambda x{:}A.M \ : \ \forall x{:}A.B(x)}$$

Shorthand: $A \to B$ is $\forall x{:}A.B$, where $x \notin FV(B)$

concrete Coq syntax:

`fun n:nat =>` M `:` `forall n:nat, vector n`

application rule:

$$\frac{\Gamma \vdash F : A \to B \quad \Gamma \vdash G : A}{\Gamma \vdash F\,G : B} \qquad \frac{\Gamma \vdash F : \forall x{:}A.B(x) \quad \Gamma \vdash G : A}{\Gamma \vdash F\,G : B[G/x]}$$

Coq — formalism: fun for all

simple types abstraction rule:

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M \ : \ A \to B}$$

dependent types abstraction rule:

$$\frac{\Gamma, x{:}A \vdash M : B(x)}{\Gamma \vdash \lambda x{:}A.M \ : \ \forall x{:}A.B(x)}$$

Shorthand: $A \to B$ is $\forall x{:}A.B$, where $x \notin FV(B)$

concrete Coq syntax:

`fun n:nat =>` M `:` `forall n:nat, vector n`

application rule:

$$\frac{\Gamma \vdash F : A \to B \quad \Gamma \vdash G : A}{\Gamma \vdash F\,G : B} \qquad \frac{\Gamma \vdash F : \forall x{:}A.B(x) \quad \Gamma \vdash G : A}{\Gamma \vdash F\,G : B[G/x]}$$

Coq — typing rules: sorts

- Sorts in Coq:

$$\begin{matrix} Prop \\ Set \end{matrix} : Type_1 : Type_2 : \ldots$$

- Cummulativity (or sub-sorting):

$$Prop \leq Set \leq Type_1 \leq Type_2 \leq \ldots$$

Coq — typing rules: sorts

- Sorts in Coq:

$$\begin{matrix} Prop \\ Set \end{matrix} : Type_1 : Type_2 : \ldots$$

- Cummulativity (or sub-sorting):

$$Prop \leq Set \leq Type_1 \leq Type_2 \leq \ldots$$

Coq — products

product rule

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \forall x{:}A.B : s_2}$$ if s_1 and s_2 satisfy ...

- $s_1 \leq s_2$, or
- $s_2 = Prop$

cummulativity rule

$$\frac{\Gamma \vdash M : s_1}{\Gamma \vdash M : s_2}$$ jeśli $s_1 \leq s_2$

Coq — products

product rule

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \forall x{:}A.B : s_2}$$ if s_1 and s_2 satisfy ...

- $s_1 \leq s_2$, or
- $s_2 = Prop$

cummulativity rule

$$\frac{\Gamma \vdash M : s_1}{\Gamma \vdash M : s_2}$$ jeśli $s_1 \leq s_2$

Coq — products

product rule

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \forall x{:}A.B : s_2}$$ if s_1 and s_2 satisfy ...

- $s_1 \leq s_2$, or
- $s_2 = \mathit{Prop}$

cummulativity rule

$$\frac{\Gamma \vdash M : s_1}{\Gamma \vdash M : s_2}$$ jeśli $s_1 \leq s_2$

Coq — reductions

- beta

  $(\lambda x{:}A.M)N \longrightarrow_\beta M[N/x]$

- eta expansion (if M is of a functional type)

  $M \longrightarrow_\eta \lambda x{:}A.Mx$

- delta

  (definition unfolding)

- zeta

  `(let x:=N in M)` $\longrightarrow_\zeta$ `M[N/x]`

- iota

  (inductive types reductions — soon :)

Coq — reductions

- beta

  $(\lambda x{:}A.M)N \longrightarrow_\beta M[N/x]$

- eta expansion (if M is of a functional type)

  $M \longrightarrow_\eta \lambda x{:}A.Mx$

- delta

  (definition unfolding)

- zeta

  `(let x:=N in M)` $\longrightarrow_\zeta$ `M[N/x]`

- iota

  (inductive types reductions — soon :)

Coq — reductions

- beta

  $(\lambda x{:}A.M)N \longrightarrow_\beta M[N/x]$

- eta expansion (if M is of a functional type)

  $M \longrightarrow_\eta \lambda x{:}A.Mx$

- delta

  (definition unfolding)

- zeta

  `(let x:=N in M)` $\longrightarrow_\zeta$ `M[N/x]`

- iota

  (inductive types reductions — soon :)

Coq — reductions

- beta

  $(\lambda x{:}A.M)N \longrightarrow_\beta M[N/x]$

- eta expansion (if M is of a functional type)

  $M \longrightarrow_\eta \lambda x{:}A.Mx$

- delta

  (definition unfolding)

- zeta

  `(let x:=N in M)` $\longrightarrow_\zeta$ `M[N/x]`

- iota

  (inductive types reductions — soon :)

Coq — reductions

- beta

  $(\lambda x{:}A.M)N \longrightarrow_\beta M[N/x]$

- eta expansion (if M is of a functional type)

  $M \longrightarrow_\eta \lambda x{:}A.Mx$

- delta

  (definition unfolding)

- zeta

  `(let x:=N in M)` $\longrightarrow_\zeta$ `M[N/x]`

- iota

  (inductive types reductions — soon :)

Coq — conversion

conversion rule

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A =_{\beta\eta\delta\zeta\iota} A' \qquad \Gamma \vdash A' : s}{\Gamma \vdash M : A'}$$

`vector nat 4` $=_{iota}$ `vector nat (2+2)`

Coq — conversion

conversion rule

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A =_{\beta\eta\delta\zeta\iota} A' \qquad \Gamma \vdash A' : s}{\Gamma \vdash M : A'}$$

$$\texttt{vector nat 4} =_{iota} \texttt{vector nat (2+2)}$$

Coq — conversion

conversion rule

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A =_{\beta\eta\delta\zeta\iota} A' \qquad \Gamma \vdash A' : s}{\Gamma \vdash M : A'}$$

$$\texttt{vector nat 4} =_{iota} \texttt{vector nat (2+2)}$$

Coq — inductive definitions

```
Inductive bool : Set :=
  true : bool
| false : bool.
```

- Inductive type:

```
bool : Set
```

- Constructors:

```
true : bool
false : bool
```

- Destruction (case analysis):

```
match n with
| true => ...
| false => ...
end
```

Coq — inductive definitions

```
Inductive bool : Set :=
  true : bool
| false : bool.
```

- Inductive type:

```
bool : Set
```

- Constructors:

```
true : bool
false : bool
```

- Destruction (case analysis):

```
match n with
| true => ...
| false => ...
end
```

Coq — inductive definitions

```
Inductive bool : Set :=
  true : bool
| false : bool.
```

- Inductive type:

```
bool : Set
```

- Constructors:

```
true : bool
false : bool
```

- Destruction (case analysis):

```
match n with
| true => ...
| false => ...
end
```

Coq — inductive definitions

```
Inductive bool : Set :=
  true : bool
| false : bool.
```

- Inductive type:

```
bool : Set
```

- Constructors:

```
true : bool
false : bool
```

- Destruction (case analysis):

```
match n with
| true => ...
| false => ...
end
```

Coq — parametric inductive definitions

```
Inductive list (A:Set): Set :=
| nil : list A
| cons : A -> list A -> list A.
```

- Parametric inductive type:

```
list : Set -> Set
```

- Constructors:

```
nil : forall A:Set, list A
cons : forall A:Set, A -> list A -> list A
```

- Structural recursion

```
Fixpoint length (A:Set)(l:list A) : nat :=
match l with
| nil => 0
| cons hd tl => S (length A tl)
end.
```

Coq — parametric inductive definitions

```
Inductive list (A:Set): Set :=
| nil : list A
| cons : A -> list A -> list A.
```

- Parametric inductive type:

```
list : Set -> Set
```

- Constructors:

```
nil : forall A:Set, list A
cons : forall A:Set, A -> list A -> list A
```

- Structural recursion

```
Fixpoint length (A:Set)(l:list A) : nat :=
match l with
| nil => 0
| cons hd tl => S (length A tl)
end.
```

Coq — parametric inductive definitions

```
Inductive list (A:Set): Set :=
| nil : list A
| cons : A -> list A -> list A.
```

- Parametric inductive type:

```
list : Set -> Set
```

- Constructors:

```
nil : forall A:Set, list A
cons : forall A:Set, A -> list A -> list A
```

- Structural recursion

```
Fixpoint length (A:Set)(l:list A) : nat :=
match l with
| nil => 0
| cons hd tl => S (length A tl)
end.
```

Coq — parametric inductive definitions

```
Inductive list (A:Set): Set :=
| nil : list A
| cons : A -> list A -> list A.
```

- Parametric inductive type:

```
list : Set -> Set
```

- Constructors:

```
nil : forall A:Set, list A
cons : forall A:Set, A -> list A -> list A
```

- Structural recursion

```
Fixpoint length (A:Set)(l:list A) : nat :=
match l with
| nil => 0
| cons hd tl => S (length A tl)
end.
```

Coq — inductive predicates

```
Inductive sorted : list Z -> Prop :=
| sorted0 : sorted nil
| sorted1: forall z:Z, sorted (z :: nil)
| sorted2: forall (z1 z2 : Z) (l : list Z),
    z1 <= z2 -> sorted (z2 :: l) -> sorted (z1 :: z2 :: l).
```

- Inductive predicate:

```
sorted : list Z -> Prop
```

- Constructors:

```
sorted0
sorted1
sorted2
```

Coq — inductive predicates

```
Inductive sorted : list Z -> Prop :=
| sorted0 : sorted nil
| sorted1: forall z:Z, sorted (z :: nil)
| sorted2: forall (z1 z2 : Z) (l : list Z),
    z1 <= z2 -> sorted (z2 :: l) -> sorted (z1 :: z2 :: l).
```

- Inductive predicate:

```
sorted : list Z -> Prop
```

- Constructors:

```
sorted0
sorted1
sorted2
```

Coq — inductive predicates

```
Inductive sorted : list Z -> Prop :=
| sorted0 : sorted nil
| sorted1: forall z:Z, sorted (z :: nil)
| sorted2: forall (z1 z2 : Z) (l : list Z),
    z1 <= z2 -> sorted (z2 :: l) -> sorted (z1 :: z2 :: l).
```

- Inductive predicate:

```
sorted : list Z -> Prop
```

- Constructors:

```
sorted0
sorted1
sorted2
```