The Coq proof assistant : principles and practice

J.-F. Monin

Université Grenoble Alpes

2016

Lecture 5

Outline

Excluded middle

Outline

Outline

Excluded middle

Discriminate

Excluded middle

A consequence of the computational reading of disjunction

Constructive (intuitionistic) logic

- $P \lor \neg P$ is not a theorem
- $\neg\neg(P \lor \neg P)$ is a theorem
- similar for $\{P\} + \{\neg P\}$

Excluded middle

A consequence of the computational reading of disjunction

Constructive (intuitionistic) logic

- $P \lor \neg P$ is not a theorem
- $\neg\neg(P \lor \neg P)$ is a theorem
- similar for $\{P\} + \{\neg P\}$

Examples

- $\forall n\ m : nat,\ \{n = m\} + \{\neg n = m\}$

Excluded middle

A consequence of the computational reading of disjunction

Constructive (intuitionistic) logic

- $P \lor \neg P$ is not a theorem
- $\neg\neg(P \lor \neg P)$ is a theorem
- similar for $\{P\} + \{\neg P\}$

Examples

- $\forall n\ m : nat,\ \{n = m\} + \{\neg n = m\}$ OK... with work

Excluded middle

A consequence of the computational reading of disjunction

Constructive (intuitionistic) logic

- $P \lor \neg P$ is not a theorem
- $\neg\neg(P \lor \neg P)$ is a theorem
- similar for $\{P\} + \{\neg P\}$

Examples

- $\forall n\ m : nat,\ \{n = m\} + \{\neg n = m\}$ OK... with work
- $\forall f : nat \to nat,\ \{\exists n,\ f\ n = 0\} + \{\forall n, \neg f\ n = 0\}$

Excluded middle

A consequence of the computational reading of disjunction

Constructive (intuitionistic) logic

- $P \lor \neg P$ is not a theorem
- $\neg\neg(P \lor \neg P)$ is a theorem
- similar for $\{P\} + \{\neg P\}$

Examples

- $\forall n\ m : nat,\ \{n = m\} + \{\neg n = m\}$ OK... with work
- $\forall f : nat \to nat,\ \{\exists n,\ f\ n = 0\} + \{\forall n, \neg f\ n = 0\}$
  just impossible

Excluded middle

A consequence of the computational reading of disjunction

Constructive (intuitionistic) logic

- $P \lor \neg P$ is not a theorem
- $\neg\neg(P \lor \neg P)$ is a theorem
- similar for $\{P\} + \{\neg P\}$

Examples

- $\forall n\ m : nat,\ \{n = m\} + \{\neg n = m\}$ OK... with work
- $\forall f : nat \to nat,\ \{\exists n,\ f\ n = 0\} + \{\forall n, \neg f\ n = 0\}$
  just impossible

Notes

- $\forall n, \neg P\ n$ is equivalent to $\neg\exists n, P\ n$
- $\forall f\ g : nat \to nat,\ f = g \lor \neg f = g$

More on Excluded middle

Admissible axioms

- $P \vee \neg P$ is admissible:
  `Require Import Classical.`
  Can be convenient, but often stronger than really needed
  Matter of taste...
- $\{P\} + \{\neg P\}$ is not admissible
  Consistent with confidentiality (see above)

Outline

Excluded middle

Discriminate

How to discriminate constructors

Aburd = extreme confusion

- A trivial consequence of absurdity (False) is that all values are equal, and that all types are equal as well (including in Set and Prop).

How to discriminate constructors

Aburd = extreme confusion

- A trivial consequence of absurdity (False) is that all values are equal, and that all types are equal as well (including in Set and Prop).
- Conversely, if all values are equal in any type, including Prop, we get `False` = `True`, i.e., `False` becomes provable

How to discriminate constructors

Aburd = extreme confusion

- A trivial consequence of absurdity (False) is that all values are equal, and that all types are equal as well (including in Set and Prop).
- Conversely, if all values are equal in any type, including Prop, we get `False` = `True`, i.e., `False` becomes provable
- Pattern-matching on constructors allows us to map distinct constructors C_i to different expressions E_i if 2 constructors C_i and C_j happened to be equated, this confusion could then be propagated to the corresponding expressions E_i and E_j; Taking $E_i =$ `True` and $E_j =$ `False`, `False` becomes provable

How to discriminate constructors

Aburd = extreme confusion

- A trivial consequence of absurdity (False) is that all values are equal, and that all types are equal as well (including in Set and Prop).
- Conversely, if all values are equal in any type, including Prop, we get `False` = `True`, i.e., `False` becomes provable
- Pattern-matching on constructors allows us to map distinct constructors C_i to different expressions E_i if 2 constructors C_i and C_j happened to be equated, this confusion could then be propagated to the corresponding expressions E_i and E_j;
  Taking $E_i = \texttt{True}$ and $E_j = \texttt{False}$, `False` becomes provable

See series5_false.v