Le Raisonnement Logique dans l'Assistant de Preuve Coq

Automates et Logiques, Licence Info, Université de Lille 1

Reynald Affeldt

National Institute of Advanced Industrial Science and Technology

25 février 2016

Objectif

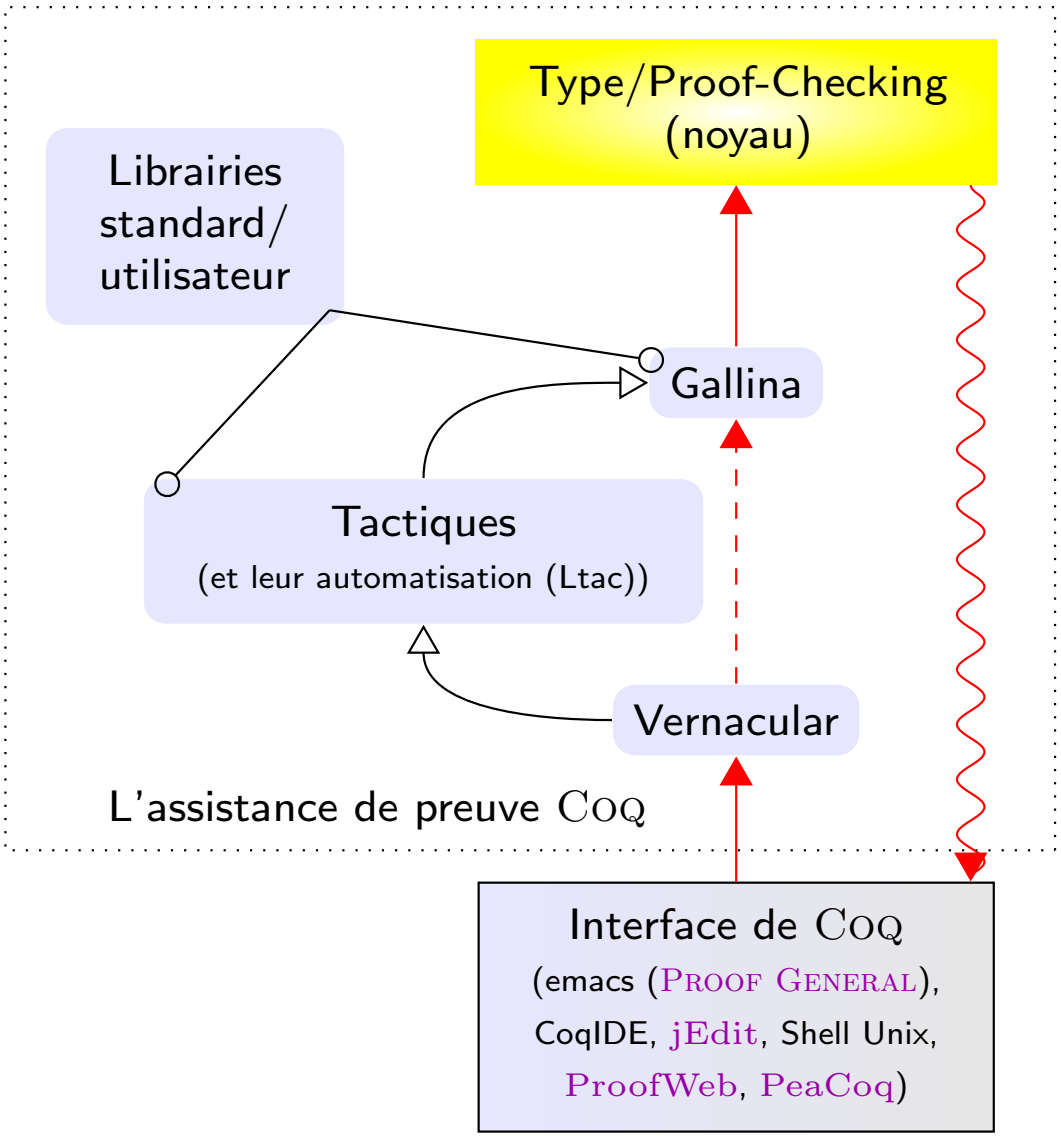
- Coq [CDT16] est une implémentation de la *théorie des types*, c'est un formalisme dans lequel la logique propositionnelle et la logique du premier ordre sont facilement représentables
- On raisonne dans Coq avec des *tactiques* [CDT16, Chapitre 8] qui ressemblent beaucoup aux règles de déduction naturelle
- On va expliquer les tactiques de Coq à la lumière des règles de déduction naturelle (telles que présentées dans [DNR03])
- Il est important de garder en tête que Coq est avant tout un λ-calcul typé : on regardent les types comme des formules de logique et les termes comme des preuves
- une formule est vraie quand on peut trouver un terme de preuve qui a le bon type ; Coq est un exemple d'application de l'*isomorphisme de Curry-Howard*

Plan

L'assistant de preuve Coq

- Développé à l'INRIA depuis 1984 à l'initiative de T. Coquand et G. Huet
- C'est essentiellement un λ-calcul typé appelé Gallina
  - le Calcul des Constructions Inductives [CP90, PM92]
  - une extension du Calcul des Constructions [CH84, CH85, CH86, CH88]
- Très utilisé
  - En recherche comme dans l'enseignement, et ce dans le monde entier
  - Prix scientifiques : ACM SIGPLAN Programming Languages Software 2013 award, ACM Software System 2013 award
  - Quelques applications pré-industrielles (certification CC EAL7)
- Success stories : le théorème des quatre couleurs [Gon05, Gon08], un compilateur C [Ler09, BL09], le théorème de l'ordre impair [GAA+13], etc.

Organisation du système Coq



→ : Ajout de structures de données et de lemmes avec le langage de commandes Vernacular

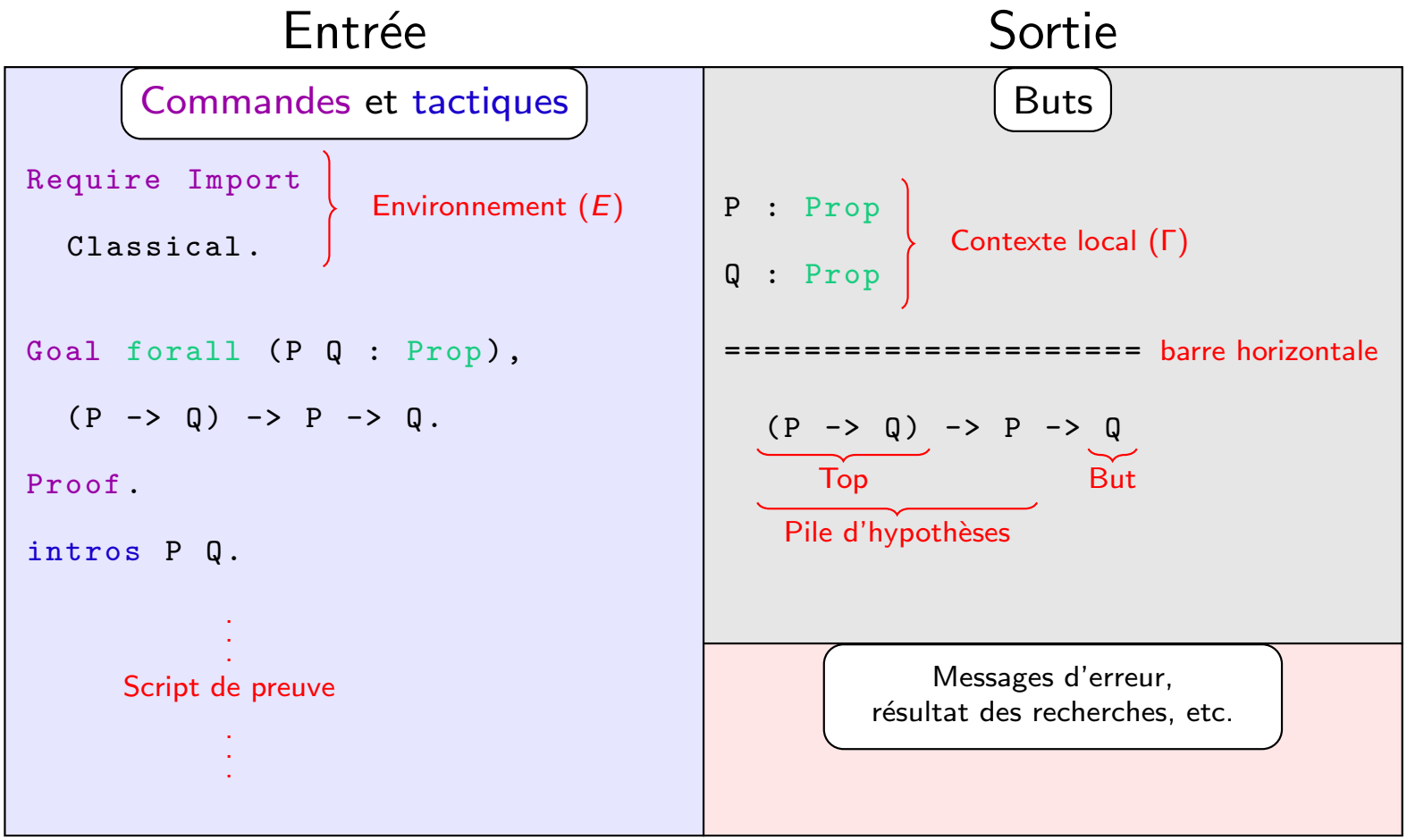
⇢ : Écriture directe de termes de preuve avec Gallina

⇝ : Messages d'informations ou d'erreurs quand le type/proof-checking échoue

⇾ : En pratique, on écrit les termes de preuve indirectement avec les tactiques

⊸ : La librarie standard propose des structures de données, des tactiques et des lemmes déjà prouvés

L'interface de Coq



Lecture de la sortie de Coq comme un séquent

- ```
  p : P
  q : Q
  ======
  R
  ```
  se lit comme $P, Q \vdash R$

- `P`, `Q`, `R` sont des propositions
- Les preuves (`p`, `q`) des hypothèses (`P`, `Q`) sont explicites
- La preuve (en cours de construction) de la conclusion `R` peut être visualisée avec la commande `Show Proof`
  ```

Plan

La règle de déduction naturelle « Axiome »

- « On peut prouver les propositions dont on dispose d'une preuve »
- Ce raisonnement est implémenté par la tactique `exact`
  - La tactique la plus basique puisqu'elle ne sert qu'à appeler le type-checker

$$\frac{}{\Gamma, A \vdash A}\ \text{ax}$$

```
A : Prop
a : A
==========                →exact a.→  No more subgoals.
A
```

Introduction de l'implication ($\to$)

- $\ll$ Pour prouver $A \to B$, il suffit de prouver B sous l'hypothèse $A \gg$
- Il s'agit juste de faire passer une hypothèse d'un côté ou de l'autre du $\vdash$ avec les tactiques `intros`/`revert`
  - Les preuves des hypothèses dans le contexte local doivent être nommées

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \to_i$$

```
A, B : Prop
==========
A -> B
```

$\to$`intros a.`[1]$\to$
$\leftarrow$`revert a.`[2]$\leftarrow$

```
A, B : Prop
a : A
==========
B
```

1. Variante limitée à un seul argument : `intro`
2. `revert` retire l'hypothèse du contexte local, `generalize` en fait une copie

Élimination de l'implication ($\rightarrow$)

- $\ll$ Sachant $A \rightarrow B$ et A, je déduis $B \gg$ (modus ponens)
- On voit l'élimination de l'implication comme une application de fonction
  - Si `ab` une preuve de `A -> B` et `a` une preuve de `A`, alors `ab a` est une preuve de `B`
- On utilise la tactique `apply` qui ne génère qu'un sous-but [3]

$$\frac{\Gamma \vdash A \rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B} \rightarrow_e$$

```
A, B : Prop                                  A : Prop
ab : A -> B                                  ab : A -> B
a : A              →apply ab.→               a : A
==========                                   ==========
B                                            A
```

3. `cut` correspond davantage à $\rightarrow_e$ mais est moins pratique

Un exemple de preuve (1/7)

Écriture du but

$$\vdash (A \to B \to C) \to (A \to B) \to A \to C$$

```
Lemma hilbertS (A B C : Prop) :
  (A -> B -> C) -> (A -> B) -> A -> C.
Proof.
```

Un exemple de preuve (2/7)

Introduction de la première implication

$$\frac{\overbrace{A \to B \to C}^{\text{abc}} \vdash (A \to B) \to A \to C}{\vdash (A \to B \to C) \to (A \to B) \to A \to C} \to_i$$

```
Lemma hilbertS (A B C : Prop) :
  (A -> B -> C) -> (A -> B) -> A -> C.
Proof.
intros abc.
```

Un exemple de preuve (3/7)

Introduction des trois implications

$$\dfrac{\dfrac{\dfrac{\overbrace{A \to B \to C}^{\texttt{abc}}, \overbrace{A \to B}^{\texttt{ab}}, \overbrace{A}^{\texttt{a}} \vdash C}{A \to B \to C, A \to B \vdash A \to C}\,{\to_i}}{A \to B \to C \vdash (A \to B) \to A \to C}\,{\to_i}}{\vdash (A \to B \to C) \to (A \to B) \to A \to C}\,{\to_i}$$

```
Lemma hilbertS (A B C : Prop) :
  (A -> B -> C) -> (A -> B) -> A -> C.
Proof.
intros abc ab a.
```

Un exemple de preuve (4/7)

Première élimination d'une implication

$$\dfrac{\dfrac{\dfrac{\dfrac{A \to B \to C, A \to B, A \vdash A \qquad A \to B \to C, A \to B, A \vdash B}{\overbrace{A \to B \to C}^{\text{abc}}, A \to B, A \vdash C}\to_e}{A \to B \to C, A \to B \vdash A \to C}\to_i}{A \to B \to C \vdash (A \to B) \to A \to C}\to_i}{\vdash (A \to B \to C) \to (A \to B) \to A \to C}\to_i$$

```
Lemma hilbertS (A B C : Prop) :
  (A -> B -> C) -> (A -> B) -> A -> C.
Proof.
intros abc ab a.
apply abc.
```

(On matche la conclusion de $A \to B \to C$ avec C et génère deux sous-buts A et B)

Un exemple de preuve (5/7)

Application de la règle ≪ Axiome ≫

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \to B \to C, A \to B, \overbrace{A}^{a} \vdash A}\,\text{ax} \qquad A \to B \to C, A \to B, A \vdash B}{A \to B \to C, A \to B, A \vdash C}\,\to_e}{A \to B \to C, A \to B \vdash A \to C}\,\to_i}{A \to B \to C \vdash (A \to B) \to A \to C}\,\to_i}{\vdash (A \to B \to C) \to (A \to B) \to A \to C}\,\to_i$$

```
Lemma hilbertS (A B C : Prop) :
  (A -> B -> C) -> (A -> B) -> A -> C.
Proof.
intros abc ab a.
apply abc.
- exact a.
```

Un exemple de preuve (6/7)

Deuxième élimination d'une implication

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \to B \to C, A \to B, A \vdash A}\,\text{ax} \qquad \dfrac{}{A \to B \to C, \overbrace{A \to B}^{\text{ab}}, A \vdash B}}{A \to B \to C, A \to B, A \vdash C}\,\to_e}{A \to B \to C, A \to B \vdash A \to C}\,\to_i}{A \to B \to C \vdash (A \to B) \to A \to C}\,\to_i}{\vdash (A \to B \to C) \to (A \to B) \to A \to C}\,\to_i$$

```
Lemma hilbertS (A B C : Prop) :
  (A -> B -> C) -> (A -> B) -> A -> C.
Proof.
intros abc ab a.
apply abc.
- exact a.
- apply ab.
```

(On matche la conclusion de $A \to B$ avec B et génère un sous-but A)

Un exemple de preuve (7/7)

Axiome et Qed

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \to B \to C, A \to B, \overbrace{A}^{a} \vdash A}\,\text{ax}}{A \to B \to C, A \to B, A \vdash B}\,{\to_e} \qquad \dfrac{}{A \to B \to C, A \to B, A \vdash A}\,\text{ax}}{A \to B \to C, A \to B, A \vdash C}\,{\to_e}}{A \to B \to C, A \to B \vdash A \to C}\,{\to_i}}{A \to B \to C \vdash (A \to B) \to A \to C}\,{\to_i}}{\vdash (A \to B \to C) \to (A \to B) \to A \to C}\,{\to_i}$$

```
Lemma hilbertS (A B C : Prop) :
  (A -> B -> C) -> (A -> B) -> A -> C.
Proof.
intros abc ab a.
apply abc.
- exact a.
- apply ab.
  exact a.
Qed.
```

Qed (Quod Erat Demonstrandum) enregistre la preuve sous le nom hilbertS

Différence entre preuve et script de preuve

- En fait, on a écrit un *script* de preuve
- La preuve en elle-même peut être observée comme suit :

```
Lemma hilbertS (A B C : Prop) :
  (A -> B -> C) -> (A -> B) -> A -> C.
Proof.
intros abc ab a.
apply abc.
- exact a.
- apply ab.
  exact a.
Show Proof.
```

```
(fun (A B C : Prop) (abc : A -> B -> C) (ab : A -> B) (a : A) =>
 abc a (ab a))
```

- Les tactiques ne sont qu'un moyen détourné pour écrire des termes du λ-calcul qui sont regardés comme des preuves
- Coq est un exemple d'implémentation de l'isomorphisme de Curry-Howard

Un Exemple de Terme de Preuve

L'axiome S de Hilbert

- On montre tous les termes de preuve en construction qui se cachent derrière les buts intermédiaires :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \mathtt{a} : A} \quad \dfrac{\dfrac{}{\Gamma \vdash \mathtt{a} : A}}{\Gamma \vdash \mathtt{ab\,a} : B}}{\Gamma^{4} \vdash \mathtt{abc\,a\,(ab\,a)} : C}}{\mathtt{abc} : A \to B \to C, \mathtt{ab} : A \to B \vdash \underset{\mathtt{abc\,a\,(ab\,a)}}{\lambda \mathtt{a} : A.} : A \to C}}{\mathtt{abc} : A \to B \to C \vdash \underset{\mathtt{abc\,a\,(ab\,a)}}{\lambda \mathtt{ab} : A \to B.\lambda \mathtt{a} : A.} : (A \to B) \to A \to C}}{\vdash \underset{\mathtt{abc\,a\,(ab\,a)}}{\lambda \mathtt{abc} : A \to B \to C.\lambda \mathtt{ab} : A \to B.\lambda \mathtt{a} : A.} : (A \to B \to C) \to (A \to B) \to A \to C}$$

- On comprend mieux pourquoi l'interface de Coq ne nous le montre pas systématiquement. . .

4. $\Gamma = \mathtt{abc} : A \to B \to C, \mathtt{ab} : A \to B, \mathtt{a} : A$

Plan

Introduction de la conjonction ($\wedge$)

- « Pour prouver $A \wedge B$, il suffit de prouver A et B »[5]
  - Création d'une branche dans un arbre de preuve
- Ce raisonnement est implémenté par la tactique `split`

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge_i$$

```
A, B : Prop                          A, B : Prop      A, B : Prop
a : A                                a : A            a : A
b : B          →split→               b : B            b : B
==========                           ==========       ==========
A /\ B                               A                B
```

5. oui, ça résonne comme une lapalissade

Élimination de la conjonction ($\wedge$)

- $\ll$ Pour prouver A (resp. B), il suffit de prouver $A \wedge B \gg$ (sic)
  - Idée : un conjonction peut être spécialisée
- Combinaison de la tactique `exact` et des lemmes `proj1`/`proj2`

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge_e^g \qquad \left(\text{resp. } \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge_e^d \right)$$

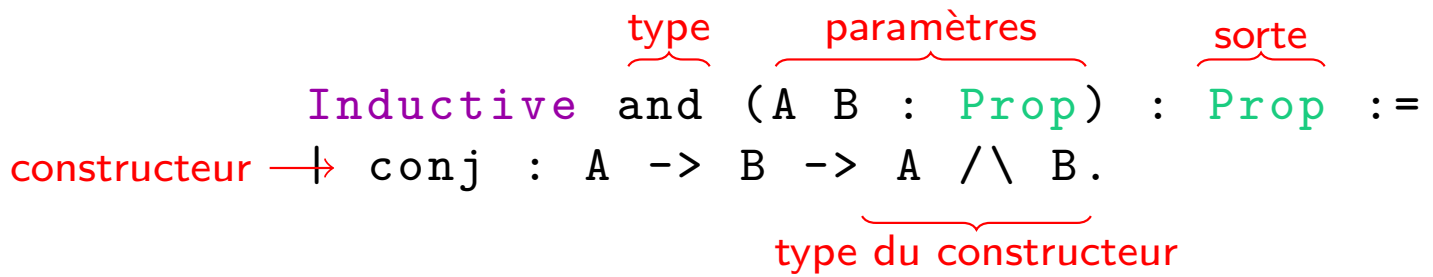
```
A, B : Prop
ab : A /\ B
==========
A
```

→`exact (proj1 ab)`[6]→ `No more subgoals.`

6. respectivement `proj2`

Comment est définie la conjonction ($\wedge$) ?

La conjonction est définie comme le type des paires de preuves :



- Soient `A` et `B` deux `Prop`ositions, alors `A /\ B` est une `Prop`osition
- `A /\ B` est une notation pour `and A B`
- Soient `a` une preuve de `A` et `b` une preuve de `B`, alors `conj a b` est une preuve de `A /\ B`

Aspects techniques de l'utilisation de la conjonction ($\wedge$)

- La tactique `split` correspond à l'application du constructeur `conj` vu comme une fonction :
  `split` $\approx$ `apply conj`
- `proj1`/`proj2` sont deux lemmes standards :

```
Theorem proj1 : A /\ B -> A.
Theorem proj2 : A /\ B -> B.
```

- En pratique, plutôt que `proj1` et `proj2`, on préfère souvent utiliser la tactique `destruct` :

```
A, B : Prop
ab : A /\ B
==========
A
```

→`destruct ab as [a b]`→

```
A, B : Prop
a : A
b : B
==========
A
```

Exemple de terme de preuve avec la conjonction

La tactique `destruct` est un moyen détourné pour faire du pattern-matching en Gallina (le langage de Coq) :

```
Lemma myproj1 (A B : Prop) : A /\ B -> A.
Proof.
exact (fun ab : A /\ B =>
  match ab with conj a b => a end).
Qed.
```

Introduction de la disjonction ($\vee$)

- $\ll$ Pour prouver $A \vee B$, il suffit de prouver A ou B $\gg$
  - Idée : on utilise cette règle pour spécialiser un but
  - Comparez avec $\wedge_e^g / \wedge_e^d$
- Ce raisonnement est implémenté par les tactiques `left` et `right`

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee_i^g \quad \left(\text{resp.} \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee_i^d \right)$$

```
A, B : Prop
a : A
==========
A \/ B
```

$\rightarrow$`left`[7]$\rightarrow$

```
A, B : Prop
a : A
==========
A
```

7. respectivement `right`

Élimination de la disjonction ($\vee$)

- Une disjunction permet de dichotomiser un raisonnement
  - Création d'une branche dans un arbre de preuve
  - Comparez avec $\wedge_i$
- On utilise la tactique `destruct`

$$\frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C} \vee_e$$

```
A,B,C : Prop
ab : A \/ B
==========
C
```

→`destruct ab as [a|b]`→

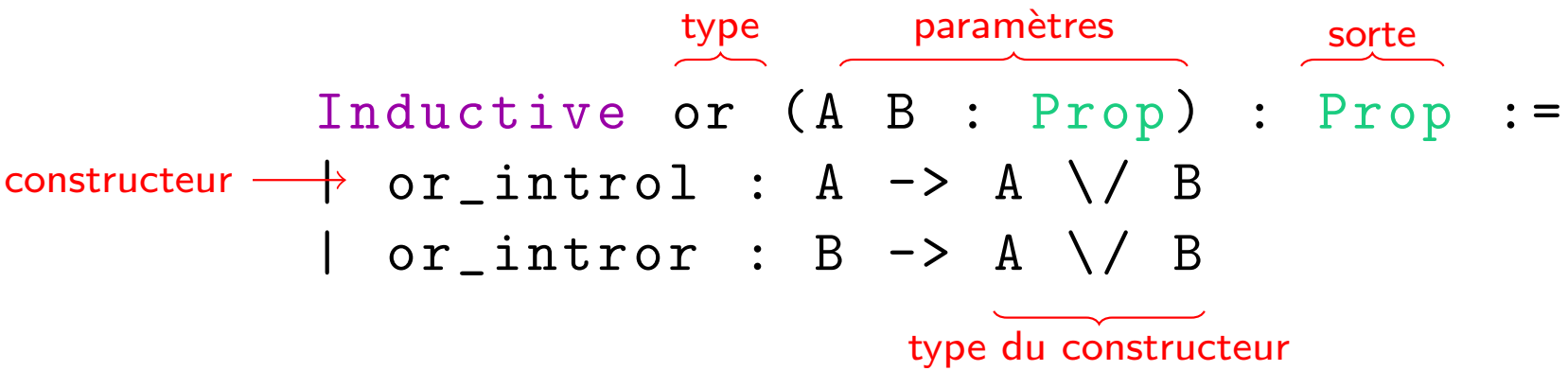
```
A,B : Prop
a : A
==========
C
```

```
A,B : Prop
a : A
==========
C
```

(`[a|b]` et non `[a b]` comme dans le cas de la conjonction—slide 25)

Comme est définie la disjonction ?

La disjonction est définie comme un type avec deux constructeurs :



- Soient `A` et `B` deux `Prop`ositions, alors `A \/ B` est une `Prop`osition
- `A \/ B` est une notation pour `or A B`
- Soit `a` une preuve de `A`, alors `or_introl B a` est une preuve de `A \/ B`
- Soit `b` une preuve de `B`, alors `or_intror A b` est une preuve de `A \/ B`

Aspects techniques de l'utilisation de la disjonction ($\vee$)

La tactique `left` correspond à l'application du constructeur `or_introl` vu comme une fonction :

$$\texttt{left} \approx \texttt{apply or\_introl}$$

La tactique `right` correspond à l'application du constructeur `or_intror` vu comme une fonction :

$$\texttt{right} \approx \texttt{apply or\_intror}$$

Un exemple de terme de preuve avec la disjonction

La règle d'élimination de la disjonction

Rappel : $$\frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C} \vee_e$$

```
Lemma or_elim (A B C : Prop) :
  (A -> C) -> (B -> C) -> A \/ B -> C.
Proof.
```

Un exemple de terme de preuve avec la disjonction

Preuve avec des tactiques

```
Lemma or_elim_tactique (A B C : Prop) :
  (A -> C) -> (B -> C) -> A \/ B -> C.
Proof.
intros ac bc.
destruct 1 as [a | b].
- apply ac.
  exact a.
- apply bc.
  exact b.
Qed.
```

Un exemple de terme de preuve avec la disjonction

Preuve avec un fonction Gallina

```
Lemma or_elim_gallina (A B C : Prop) :
  (A -> C) -> (B -> C) -> A \/ B -> C.
Proof.
exact (fun (ac : A -> C) (bc : B -> C) (ab : A \/ B) =>
match ab with
| or_introl a => ac a
| or_intror b => bc b
end).
Qed.
```

En réalité, ces deux scripts produisent exactement la même preuve.

Plan

Le faux et la négation dans Coq

- le faux (notation papier : $\bot$) s'écrit `False` dans Coq
  - `False` n'est pas un objet de base, il est défini (slide 39)
- la négation de A (notation paper : $\neg A$) s'écrit `~ A` (lire « tilde A ») dans Coq
  - `~` n'est pas une connective de base, elle est définie :
    - $\neg A$ est défini (sur papier) comme $A \to \bot$
    - dans Coq, $A \to \bot$ devient `A -> False`
    - on note `~ A` pour `A -> False` pour simplifier la présentation

Introduction et élimination de la négation ($\neg$)

Rappel du cours [Pie16] :

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \neg_i \qquad \frac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash \bot}\ \neg_e$$

On remplace $\neg A$ par sa définition $A \to \bot$ dans les règles ci-dessus :

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash A \to \bot}\ \neg_i \qquad \frac{\Gamma \vdash A \to \bot \qquad \Gamma \vdash A}{\Gamma \vdash \bot}\ \neg_e$$

On s'aperçoit que les règles pour la négation sont une instance des règles pour l'implication (remplacer B par $\bot$ ci-dessous) :

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ \to_i \qquad \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \to_e$$

On n'a donc pas à ajouter de tactiques dans Coq pour gérer la négation

Introduction de la négation ($\neg$) dans Coq

« pour prouver $\neg A$, il suffit de prouver $\bot$ dans un contexte où A est vrai »

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \neg_i$$

```
A : Prop
==========
~ A
```

→`intros a.`→

```
A : Prop
a : A
==========
False
```

Élimination de la négation ($\neg$) dans Coq

« pour prouver $\bot$, il suffit de trouver A tel que A et $\neg A$ soient vraies »

$$\frac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash \bot}\ \neg_e$$

```
A : Prop
na : ~ A
a : A                →apply na.[8]→   No more subgoals.
==========
False
```

8. Ou bien `exact (na a)`

Comment est défini faux ($\bot$) dans Coq ?

- $\bot$ est une proposition dont on ne peut construire de preuve
- dans Coq, $\bot$ devient donc un type dont on ne peut construire les objets
- concrètement, `False` est un type inductif sans constructeur défini par la commande suivante :

Type Sort

```
Inductive False : Prop := .
```

La règle d'élimination du faux ($\bot$)

- en pratique, quand on a $\bot$ dans le contexte, on « prouve » le but en l'éliminant
- *ex falso quodlibet* : à partir de faux, on prouve n'importe quoi

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ \bot_e$$

```
A : Prop
abs : False
==========   →destruct abs.→   No more subgoals.
A
```

La logique classique dans Coq

- Coq est un formalisme *constructif* : on s'interdit par défaut les propriétés de la logique classique
  - exemple : le *tiers exclus* (≪ une proposition est nécessairement vraie ou bien fausse ≫)
- néanmoins, admettre la logique classique n'entraîne pas de contradiction
  - Coq fournit des caractérisations de la logique classique sous forme d'*axiomes* (des lemmes sans preuve)

Exemple de raisonnement classique ≪ prouvable ≫ :

≪ si à partir de $\neg A$ on peux prouver $\bot$, alors A est vraie ≫

```
Require Import Classical.

Lemma bottom_c (A : Prop) : ((~A) -> False) -> A.
Proof. ... Qed.
```

L'absurdité classique

On retrouve la règle de l'absurdité classique en utilisant le lemma `bottom_c` du slide précédent (slide 41) :

$$\frac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}\ \bot_c$$

```
A : Prop
==========
A
```
→
```
apply bottom_c.
intros na.
```
→
```
A : Prop
na : ~ A
==========
False
```

La règle de déduction naturell « Affaiblissement »

Par souci d'exhaustivité...

- Certaines hypothèses peuvent être inutiles
- Ce raisonnement est implémenté par la tactique `clear`

$$\frac{\Gamma \vdash A}{\Gamma, B \vdash A}\ \text{aff}$$

```
A, B : Prop
a : A
b : B
==========
A
```

→`clear b.`→

```
A, B : Prop
a : A
==========
A
```

Plan

Introduction du quantificateur universel

- « Pour prouver $\forall x.A$, il suffit de prouver A pour un x quelconque »
- Bonnes nouvelles :
  - $\forall(x : X), A$ n'est qu'une généralisation de $X \to A$ (même tactique)
  - Coq s'occupe de vérifier les problèmes de nommage

$$\frac{\Gamma \vdash A \qquad x \text{ n'est pas libre dans les formules de } \Gamma}{\Gamma \vdash \forall x.A}\ \forall_i$$

```
X : Type
A : X -> Prop
x : X
==========
forall x : X, A x
```

→`intros x0.`→
←`revert x0.`[9]←

```
X : Type
A : X -> Prop
x : X
x0 : X
==========
A x0
```

9. `generalize x0` crée une nouvelle variable `x1`

Élimination du quantificateur universel

- Regarder l'élimination du quantificateur universel comme l'application d'une fonction
- Similaire à l'élimination de l'implication (même tactique)

$$\frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[x := t]} \forall_e$$

```
X : Type
A : X -> Prop
Ax : forall x : X, A x
t : X
==========
A t
```

→`apply Ax.`[10]→

```
No more
  subgoals.
```

10. Ou bien `exact (Ax t)`

Introduction du quantificateur existentiel

- « Pour prouver $\exists x.A$, il suffit de trouve un témoin t tel que $A\,t$ soit vrai »
- Ce raisonnement est implémenté par la tactique `exists`

$$\frac{\Gamma \vdash A[x := t]}{\Gamma \vdash \exists x.A}\ \exists_i$$

```
X : Type                                    X : Type
x : X                                       x : X
A : X -> Prop                               A : X -> Prop
t : X                    →exists t→         t : X
At : A t                                    At : A t
==========                                  ==========
exists x0 : X, A x0                         A t
```

Élimination du quantificateur existentiel

- Regarder $\exists x.A$ comme la paire d'un témoin t et d'une preuve de $A\,t$
- La tactique `destruct` révèle le témoin et la preuve correspondante
  - Coq se charge de vérifier que le nom du témoin ne cause pas de conflit

$$\frac{\Gamma \vdash \exists x.A \qquad \Gamma, A \vdash C \quad x \text{ n'est libre ni dans les formules de } \Gamma, \text{ ni dans } C}{\Gamma \vdash C}\ \exists_e$$

```
C : Prop
X : Type
x : X
A : X -> Prop
At : exists t : X,
       A t
==========
C
```

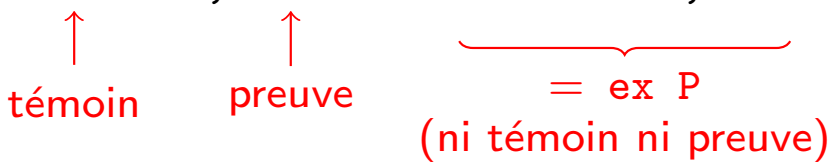
→`destruct At as [t At]`→

```
C : Prop
X : Type
x : X
A : X -> Prop
t : X
At : A t
==========
C
```

Comme est défini le quantificateur existentiel ?

Le quantificateur existentiel est implémenté comme une paire :

```
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
| ex_intro : forall x : A, P x -> exists x, P x
```



- `exists x, P x` est une notation pour `ex P`
- La tactique `exists t` n'est que l'application du constructeur : `apply (ex_intro _ t)`

Introduction de l'égalité

- La tactique de la `reflexivity` implémente le fait que l'égalité est réflexive

$$\frac{}{\Gamma \vdash t = t}\ =_i$$

```
X : Type
t : X
==========
t = t
```

→`reflexivity`→

```
No more subgoals.
```

Élimination de l'égalité

- L'élimination de l'égalité est implémentée comme une réécriture

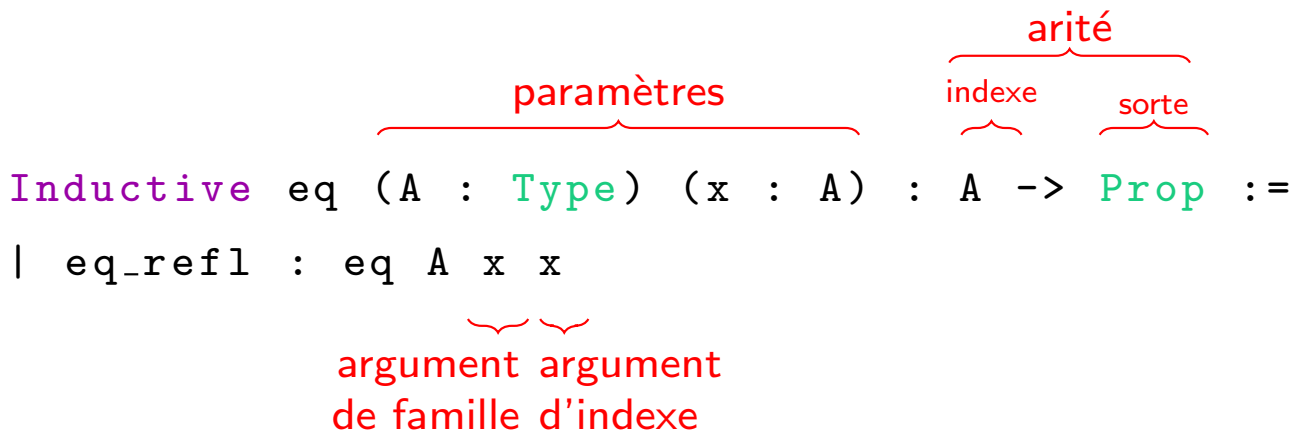
$$\frac{\Gamma \vdash A[x := t] \qquad \Gamma \vdash t = u}{\Gamma \vdash A[x := u]} =_e$$

```
X : Type                                  X : Type
t, u : X                                  t, u : X
A : X -> Prop                             A : X -> Prop
At : A t          →rewrite <-tu[11]→      At : A t
tu : t = u                                tu : t = u
==========                                ==========
A u                                       A t
```

11. Et dans l'autre sens : `rewrite ->tu` ou plus simplement `rewrite tu`

Comment est définie l'égalité ?

L'égalité est définie dans Coq comme un type inductif indexé :



- `x = y` est une notation pour `eq _ x y`
- `reflexivity` = `apply eq_refl`

Que fait `rewrite` ?

Exemple de principe d'induction

- Au moment de la définition d'un type inductive, un principe d'induction est généré (avec sa preuve !) :

```
eq_ind : forall (A : Type) (x : A) (P : A -> Prop),
  P x -> forall y : A, x = y -> P y
```

- `eq_ind` est essentiellement une fonction qui transforme `P x` en `P y` pour n'importe quel `P` (on parle de l'*égalité de Leibniz*)
- En fait, dans l'exemple précédent, `rewrite <-tu` est équivalent à :

```
apply (eq_ind _ _ At _ tu).
```

- Au final, les tactiques qui comptent vraiment dans ce cours : `intros`/`revert`, `apply`, `destruct ... as [...]` (, `clear`)

Plan

À propos des travaux pratiques

- On va reproduire des exemples des cours précédents [Pie16] et les premiers exemples de [DNR03]
- On va regarder un encodage alternatif des connectives logiques qui n'utilise pas les types inductifs (donc CoC au lieu de CIC)
- Slides et TP en ligne : `https://staff.aist.go.jp/reynald.affeldt/coq/`

Index des tactiques vues dans ce document

Bibliographie I


Sandrine Blazy and Xavier Leroy, *Mechanized semantics for the Clight subset of the C language*, J. Autom. Reasoning **43** (2009), no. 3, 263–288.


The Coq Development Team, *The Coq proof assistant reference manual*, INRIA, 2016, Version 8.5.


Thierry Coquand and Gérard Huet, *A theory of constructions (preliminary version)*, International Symposium on Semantics of Data Types, Sophia-Antipolis, 1984, Jun. 1984.


———, *Constructions : A higher order proof system for mechanizing mathematics*, Proceedings of the European Conference on Computer Algebra, Linz, Austria EUROCAL 85, April 1–3, 1985, Linz, Austria, vol. 1 (Invited Lectures), Apr. 1985, pp. 151–184.


———, *Concepts mathématiques et informatiques formalisés dans le calcul des constructions*, Tech. Report 515, INRIA Rocquencourt, Apr. 1986.


———, *The calculus of constructions*, Information and Computation **76** (1988), 95–120.


Thierry Coquand and Christine Paulin, *Inductively defined types*, Proceedings of the International Conference on Computer Logic (COLOG-88), Tallinn, USSR, December 1988, Lecture Notes in Computer Science, vol. 417, Springer, 1990, pp. 50–66.


René David, Karim Nour, and Christophe Raffalli, *Introduction à la logique*, 2ème ed., Dunod, 2003.


Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry, *A machine-checked proof of the odd order theorem*, Proceedings of the 4th International Conference on Interactive Theorem Proving, ITP 2013, Rennes, France, July 22–26, 2013, Lecture Notes in Computer Science, vol. 7998, Springer, 2013, pp. 163–179.

Bibliographie II


Georges Gonthier, *A computer-checked proof of the four colour theorem*, Tech. report, Microsoft Research, Cambridge, 2005, Available at : `http://research.microsoft.com/en-us/um/people/gonthier/4colproof.pdf`. Last access : 2014/08/04.


_____, *Formal proof—the four-color theorem*, Notices of the American Mathematical Society **55** (2008), no. 11, 1382–1393.


Xavier Leroy, *A formally verified compiler back-end*, J. Autom. Reasoning **43** (2009), no. 4, 363–446.


Thomas Pietrzak, *Logique—logique propositionnelle—*, Licence Informatique, Université de Lille 1 Sciences et Technologies, 2016, `http://www.thomaspietrzak.com/download.php?f=CoursLogique0.pdf`.


Christine Paulin-Mohring, *Inductive definitions in the sytem coq rules and properties*, Tech. Report 92–49, LIP, École Normales Supérieure de Lyon, Dec. 1992.