

Master Parisien de Recherche en Informatique  
cours 2-7-2 : assistants de preuve

Preuve de programmes fonctionnels (4)

Jean-Christophe Filliâtre

2005–2006

## Plan

- ① Principe de Poincaré
- ② Décision d'un prédicat
  - exemple : test de primalité
- ③ Raisonnements algébriques
  - exemple : associativité
- ④ Approche mixte : les traces

**idée** : remplacer des étapes de preuve par du **calcul**

c'est l'application du **principe de Poincaré**

Henri Poincaré, à propos de la preuve de  $2 + 2 = 4$  :

*« Ce n'est pas une démonstration proprement dite, [...] c'est une vérification. [...] La vérification diffère précisément de la véritable démonstration, parce qu'elle est purement analytique et parce qu'elle est stérile. »*

*(La Science et l'Hypothèse, 1902)*

**dit autrement** : il y a dans Coq un langage de programmation, utilisons-le pour programmer des méthodes de raisonnement (procédures de décision, fonctions de simplification, etc.)

⇒ on remplace de nombreuses tactiques (donc des termes de preuve) par un **test de convertibilité**

⇒ terme de preuve **plus petit** (gain de place et parfois même de temps quant à sa vérification)

**dit autrement** : il y a dans Coq un langage de programmation, utilisons-le pour programmer des méthodes de raisonnement (procédures de décision, fonctions de simplification, etc.)

⇒ on remplace de nombreuses tactiques (donc des termes de preuve) par un **test de convertibilité**

⇒ terme de preuve **plus petit** (gain de place et parfois même de temps quant à sa vérification)

**dit autrement** : il y a dans Coq un langage de programmation, utilisons-le pour programmer des méthodes de raisonnement (procédures de décision, fonctions de simplification, etc.)

⇒ on remplace de nombreuses tactiques (donc des termes de preuve) par un **test de convertibilité**

⇒ terme de preuve **plus petit** (gain de place et parfois même de temps quant à sa vérification)

c'est la règle de **conversion**

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash B : s \quad A =_{\beta\delta\iota\zeta} B}{\Gamma \vdash t : B} (\text{CONV})$$



# Règle de conversion : exemple

**Theorem** plus\_assoc :  $\forall x y z, x+(y+z)=(x+y)+z$ .

**Proof.**

```
induction x; intros.
```

le premier but est

2 subgoals

y : nat

z : nat

----- (1/2)  
 $0 + (y + z) = (0 + y) + z$

et peut être directement prouvé par

```
exact (refl_equal (y+z)).
```

# Règle de conversion : exemple

**Theorem** plus\_assoc :  $\forall x y z, x+(y+z)=(x+y)+z$ .

**Proof.**

```
induction x; intros.
```

le premier but est

2 subgoals

y : nat

z : nat

----- (1/2)  
 $0 + (y + z) = (0 + y) + z$

et peut être directement prouvé par

```
exact (refl_equal (y+z)).
```

# Règle de conversion : exemple

**Theorem** plus\_assoc :  $\forall x y z, x+(y+z)=(x+y)+z$ .

**Proof.**

```
induction x; intros.
```

le premier but est

2 subgoals

y : nat

z : nat

----- (1/2)  
 $0 + (y + z) = (0 + y) + z$

et peut être directement prouvé par

```
exact (refl_equal (y+z)).
```

## Règle de conversion : exemple

$$\Gamma \vdash \text{refl\_equal } (y + z) : y + z = y + z$$
$$\Gamma \vdash 0 + (y + z) = (0 + y) + z : \text{Prop}$$
$$\frac{y + z = y + z =_{\beta\delta\iota\zeta} 0 + (y + z) = (0 + y) + z}{\Gamma \vdash \text{refl\_equal } (y + z) : 0 + (y + z) = (0 + y) + z}$$

deux classes de problèmes où la réflexion calculatoire s'applique :

- un prédicat  $C : T \rightarrow \text{Prop}$  que l'on souhaite décider
- raisonnements algébriques de « type réécriture »

deux classes de problèmes où la réflexion calculatoire s'applique :

- un prédicat  $C : T \rightarrow \text{Prop}$  que l'on souhaite décider
- raisonnements algébriques de « type réécriture »

deux classes de problèmes où la réflexion calculatoire s'applique :

- un prédicat  $C : T \rightarrow \text{Prop}$  que l'on souhaite décider
- raisonnements algébriques de « type réécriture »

# Décision d'un prédicat $C$

soit un prédicat  $C : T \rightarrow \text{Prop}$

on définit une fonction  $f : T \rightarrow \text{bool}$  et on montre le théorème

**Theorem** `f_correct` :  $\forall x, f\ x = \text{true} \rightarrow C\ x$ .

si  $f$  est définie de telle manière que  $f\ t$  se réduise vers `true` alors on a la preuve suivante de  $C\ t$  :

`f_correct t (refl_equal true) : C t`

la taille du terme de preuve ne dépend que de  $t$ , pas de la difficulté à prouver  $t$



# Décision d'un prédicat $C$

soit un prédicat  $C : T \rightarrow \text{Prop}$

on définit une fonction  $f : T \rightarrow \text{bool}$  et on montre le théorème

**Theorem** `f_correct` :  $\forall x, f\ x = \text{true} \rightarrow C\ x$ .

si  $f$  est définie de telle manière que  $f\ t$  se réduise vers `true` alors on a la preuve suivante de  $C\ t$  :

`f_correct t (refl_equal true) : C t`

la taille du terme de preuve ne dépend que de  $t$ , pas de la difficulté à prouver  $t$

# Décision d'un prédicat $C$

soit un prédicat  $C : T \rightarrow \text{Prop}$

on définit une fonction  $f : T \rightarrow \text{bool}$  et on montre le théorème

**Theorem** `f_correct` :  $\forall x, f\ x = \text{true} \rightarrow C\ x$ .

si  $f$  est définie de telle manière que  $f\ t$  se réduise vers `true` alors on a la preuve suivante de  $C\ t$  :

`f_correct t (refl_equal true) : C t`

la taille du terme de preuve ne dépend que de  $t$ , pas de la difficulté à prouver  $t$

une tactique pour déterminer si un entier est premier

ne s'applique que pour un terme  $t$  **clos**  
(sinon,  $f t$  ne se réduit pas jusqu'à `true` mais reste bloqué sur une variable)

une solution sera apportée plus loin

ne s'applique que pour un terme  $t$  **clos**  
(sinon,  $f t$  ne se réduit pas jusqu'à `true` mais reste bloqué sur une variable)

une solution sera apportée plus loin

on souhaite effectuer des raisonnements algébriques sur des termes d'un type  $T$  (preuves d'égalité et/ou simplifications modulo commutativité, associativité, règles de réécriture, etc.)

on introduit un **domaine abstrait**  $A$  et une fonction  $i : A \rightarrow T$  pour relier le domaine abstrait au domaine concret  $T$

puis on définit une fonction  $f : A \rightarrow A$  travaillant sur le domaine abstrait dont on montre la propriété suivante :

**Theorem**  $f\_ident : \forall x, i (f x) = i x.$

on souhaite effectuer des raisonnements algébriques sur des termes d'un type  $T$  (preuves d'égalité et/ou simplifications modulo commutativité, associativité, règles de réécriture, etc.)

on introduit un **domaine abstrait**  $A$  et une fonction  $i : A \rightarrow T$  pour relier le domaine abstrait au domaine concret  $T$

puis on définit une fonction  $f : A \rightarrow A$  travaillant sur le domaine abstrait dont on montre la propriété suivante :

**Theorem** `f_ident` :  $\forall x, i (f x) = i x.$

on souhaite effectuer des raisonnements algébriques sur des termes d'un type  $T$  (preuves d'égalité et/ou simplifications modulo commutativité, associativité, règles de réécriture, etc.)

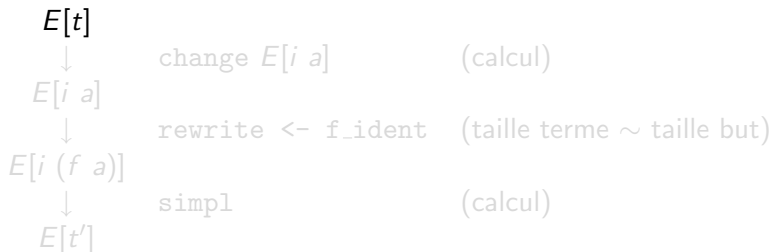
on introduit un **domaine abstrait**  $A$  et une fonction  $i : A \rightarrow T$  pour relier le domaine abstrait au domaine concret  $T$

puis on définit une fonction  $f : A \rightarrow A$  travaillant sur le domaine abstrait dont on montre la propriété suivante :

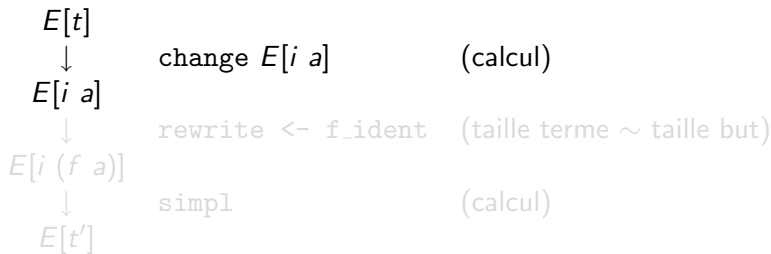
**Theorem** `f_ident` :  $\forall x, i (f x) = i x.$



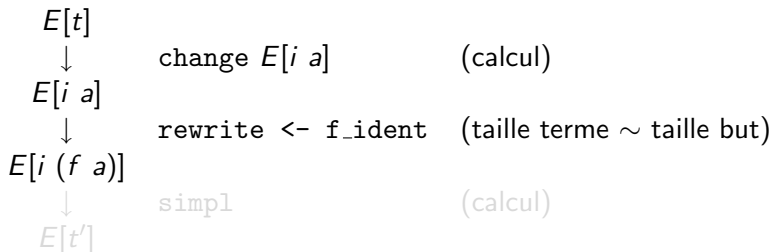
# Application : simplification



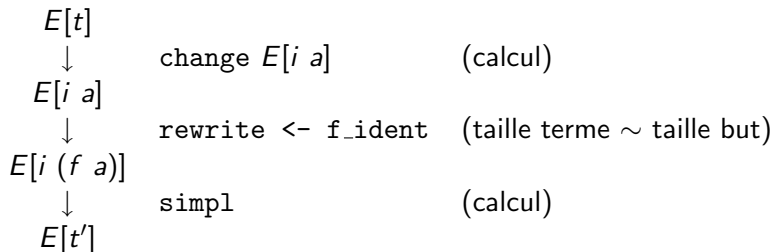
# Application : simplification



# Application : simplification



# Application : simplification



# Application : preuve d'égalité

$t_1 = t_2$		
↓	change $i\ a_1 = i\ a_2$	(calcul)
$i\ a_1 = i\ a_2$		
↓	rewrite $\leftarrow (f\_ident\ a_1)$	(taille terme
	rewrite $\leftarrow (f\_ident\ a_2)$	$\sim$ taille but)
$i\ (f\ a_1) = i\ (f\ a_2)$		
↓	reflexivity	(calcul)
goal proved		

# Application : preuve d'égalité

$t_1 = t_2$		
↓		
$i\ a_1 = i\ a_2$	change $i\ a_1 = i\ a_2$	(calcul)
	rewrite $\leftarrow$ (f_ident $a_1$ )	(taille terme
↓	rewrite $\leftarrow$ (f_ident $a_2$ )	~ taille but)
$i\ (f\ a_1) = i\ (f\ a_2)$		
↓	reflexivity	(calcul)
goal proved		

# Application : preuve d'égalité

$t_1 = t_2$		
↓	change $i a_1 = i a_2$	(calcul)
$i a_1 = i a_2$		
↓	rewrite $\leftarrow$ (f_ident $a_1$ )	(taille terme
	rewrite $\leftarrow$ (f_ident $a_2$ )	$\sim$ taille but)
$i (f a_1) = i (f a_2)$		
↓	reflexivity	(calcul)
goal proved		

# Application : preuve d'égalité

$t_1 = t_2$		
↓	change $i a_1 = i a_2$	(calcul)
$i a_1 = i a_2$		
↓	rewrite $\leftarrow$ (f_ident $a_1$ )	(taille terme
	rewrite $\leftarrow$ (f_ident $a_2$ )	$\sim$ taille but)
$i (f a_1) = i (f a_2)$		
↓	reflexivity	(calcul)
goal proved		



l'étape qui consiste à remplacer  $t$  par  $i$   $a$  impose de construire la forme abstraite  $a$ ; on appelle cela la **réification**

elle peut être réalisée

- à la main
- par une tactique écrite Ltac
- par un morceau de code ML

l'étape qui consiste à remplacer  $t$  par  $i$   $a$  impose de construire la forme abstraite  $a$ ; on appelle cela la **réification**

elle peut être réalisée

- à la main
- par une tactique écrite Ltac
- par un morceau de code ML

l'étape qui consiste à remplacer  $t$  par  $i$   $a$  impose de construire la forme abstraite  $a$ ; on appelle cela la **réification**

elle peut être réalisée

- à la main
- par une tactique écrite Ltac
- par un morceau de code ML

l'étape qui consiste à remplacer  $t$  par  $i$   $a$  impose de construire la forme abstraite  $a$ ; on appelle cela la **réification**

elle peut être réalisée

- à la main
- par une tactique écrite Ltac
- par un morceau de code ML

pour appliquer la technique dans le cas de termes **non clos** on considère des structures abstraites avec des **variables**

on paramètre la fonction d'interprétation  $i$  par une substitution de ces variables (un environnement); on a alors

**Theorem**  $f\_ident : \forall x, \forall s, i\ s\ (f\ a) = i\ s\ a.$

on peut en profiter pour remplacer les sous-termes non traitables par des variables (**abstraction**)

pour appliquer la technique dans le cas de termes **non clos** on considère des structures abstraites avec des **variables**

on paramètre la fonction d'interprétation  $i$  par une substitution de ces variables (un environnement); on a alors

**Theorem**  $f\_ident$  :  $\forall x, \forall s, i \ s \ (f \ a) = i \ s \ a.$

on peut en profiter pour remplacer les sous-termes non traitables par des variables (**abstraction**)

pour appliquer la technique dans le cas de termes **non clos** on considère des structures abstraites avec des **variables**

on paramètre la fonction d'interprétation  $i$  par une substitution de ces variables (un environnement); on a alors

**Theorem**  $f\_ident : \forall x, \forall s, i\ s\ (f\ a) = i\ s\ a.$

on peut en profiter pour remplacer les sous-termes non traitables par des variables (**abstraction**)

preuves modulo associativité

on veut simplifier des expressions contenant 0 et plus en utilisant

- $0 + x = x + 0 = x$
- l'associativité de plus



- **Ring** : simplification et preuve d'égalité dans les anneaux
  - `Ring t` : simplifie le terme  $t$
  - `Ring` : prouve un but de la forme  $t_1 = t_2$
  - `Add Ring ...` permet de déclarer de nouveaux anneaux
  
- **Field** : preuve d'égalité dans les corps commutatifs
  - `Field` : prouve un but de la forme  $t_1 = t_2$
  - peut produire des conditions de bord à prouver
  - `Add Field ...` permet de déclarer de nouveaux corps

- **Ring** : simplification et preuve d'égalité dans les anneaux
  - `Ring t` : simplifie le terme  $t$
  - `Ring` : prouve un but de la forme  $t_1 = t_2$
  - `Add Ring ...` permet de déclarer de nouveaux anneaux
  
- **Field** : preuve d'égalité dans les corps commutatifs
  - `Field` : prouve un but de la forme  $t_1 = t_2$
  - peut produire des conditions de bord à prouver
  - `Add Field ...` permet de déclarer de nouveaux corps

il n'est pas toujours facile de programmer dans Coq des procédures de décision complexes

il existe une approche **mixte** qui consiste à

- ① écrire une procédure de décision en ML qui produit **une trace**
- ② internaliser dans Coq la notion de trace et sa correction

la preuve vérifiée par Coq effectue un calcul sur la trace et sa taille est **proportionnelle à cette trace**

il n'est pas toujours facile de programmer dans Coq des procédures de décision complexes

il existe une approche **mixte** qui consiste à

- ① écrire une procédure de décision en ML qui produit **une trace**
- ② internaliser dans Coq la notion de trace et sa correction

la preuve vérifiée par Coq effectue un calcul sur la trace et sa taille est **proportionnelle à cette trace**

il n'est pas toujours facile de programmer dans Coq des procédures de décision complexes

il existe une approche **mixte** qui consiste à

- ① écrire une procédure de décision en ML qui produit **une trace**
- ② internaliser dans Coq la notion de trace et sa correction

la preuve vérifiée par Coq effectue un calcul sur la trace et sa taille est **proportionnelle à cette trace**

il n'est pas toujours facile de programmer dans Coq des procédures de décision complexes

il existe une approche **mixte** qui consiste à

- ① écrire une procédure de décision en ML qui produit **une trace**
- ② internaliser dans Coq la notion de trace et sa correction

la preuve vérifiée par Coq effectue un calcul sur la trace et sa taille est **proportionnelle à cette trace**

- **ROmega** : version réflexive de la tactique omega
  - décision dans l'arithmétique de Prestburger
  - code de recherche de preuve écrit en OCaml
  - preuve ensuite traduite dans un type Coq
  
- **Interface Elan - Coq**
  - utilisation de l'outil de réécriture Elan dans Coq

- **ROmega** : version réflexive de la tactique omega
  - décision dans l'arithmétique de Prestburger
  - code de recherche de preuve écrit en OCaml
  - preuve ensuite traduite dans un type Coq
  
- **Interface Elan - Coq**
  - utilisation de l'outil de réécriture Elan dans Coq



# Le théorème des 4 couleurs

*tout graphe planaire est 4-coloriable*

prouvé en 1976 par Appel et Haken, mais preuve **controversée** car impliquant des programmes informatiques pour traiter une analyse par cas, finie mais gigantesque

preuve simplifiée en 1995 par Robertson, Sanders, Seymour et Thomas ; moins de cas mais toujours des programmes ( $\sim 2$  heures de calcul)

preuve entièrement formalisée dans Coq 7.3.1 en 2004 par G. Gonthier (INRIA, aujourd'hui Microsoft)

les programmes font partie intégrante de la preuve ; c'est l'exemple ultime de la **preuve par réflexion**

*tout graphe planaire est 4-coloriable*

prouvé en 1976 par Appel et Haken, mais preuve **controversée** car impliquant des programmes informatiques pour traiter une analyse par cas, finie mais gigantesque

preuve simplifiée en 1995 par Robertson, Sanders, Seymour et Thomas ; moins de cas mais toujours des programmes ( $\sim 2$  heures de calcul)

preuve entièrement formalisée dans Coq 7.3.1 en 2004 par G. Gonthier (INRIA, aujourd'hui Microsoft)  
les programmes font partie intégrante de la preuve ; c'est l'exemple ultime de la **preuve par réflexion**

*tout graphe planaire est 4-coloriable*

prouvé en 1976 par Appel et Haken, mais preuve **controversée** car impliquant des programmes informatiques pour traiter une analyse par cas, finie mais gigantesque

preuve simplifiée en 1995 par Robertson, Sanders, Seymour et Thomas ; moins de cas mais toujours des programmes ( $\sim 2$  heures de calcul)

preuve entièrement formalisée dans Coq 7.3.1 en 2004 par G. Gonthier (INRIA, aujourd'hui Microsoft)

les programmes font partie intégrante de la preuve ; c'est l'exemple ultime de la **preuve par réflexion**

# Le théorème des 4 couleurs

*tout graphe planaire est 4-coloriable*

prouvé en 1976 par Appel et Haken, mais preuve **controversée** car impliquant des programmes informatiques pour traiter une analyse par cas, finie mais gigantesque

preuve simplifiée en 1995 par Robertson, Sanders, Seymour et Thomas ; moins de cas mais toujours des programmes ( $\sim$  2 heures de calcul)

preuve entièrement formalisée dans Coq 7.3.1 en 2004 par G. Gonthier (INRIA, aujourd'hui Microsoft)

les programmes font partie intégrante de la preuve ; c'est l'exemple ultime de la **preuve par réflexion**

# Le théorème des 4 couleurs

*tout graphe planaire est 4-coloriable*

prouvé en 1976 par Appel et Haken, mais preuve **controversée** car impliquant des programmes informatiques pour traiter une analyse par cas, finie mais gigantesque

preuve simplifiée en 1995 par Robertson, Sanders, Seymour et Thomas ; moins de cas mais toujours des programmes ( $\sim$  2 heures de calcul)

preuve entièrement formalisée dans Coq 7.3.1 en 2004 par G. Gonthier (INRIA, aujourd'hui Microsoft)

les programmes font partie intégrante de la preuve ; c'est l'exemple ultime de la **preuve par réflexion**