Introduction générale

Christine Paulin

Plan

- Objectifs.
- Architecture.
- Applications.
- Autres outils.

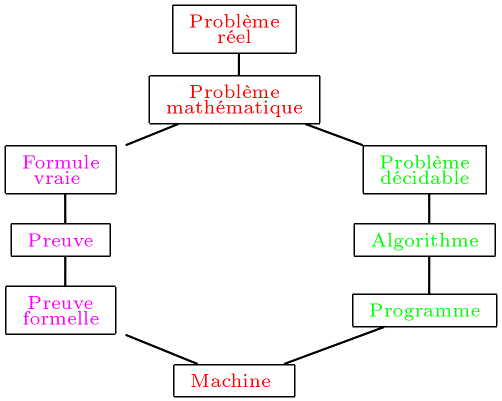
Objectifs

- Remplacer le papier et le crayon pour développer des théories.
- Traiter les modélisations informatiques.

Intérêt

- Spécification formelle.
- Standard de correction.
- Mécanisation des preuves.
- Bibliothèques.
- Évolution.

Coq dans le processus de modélisation



Assistant à la formalisation

Présentation de théories

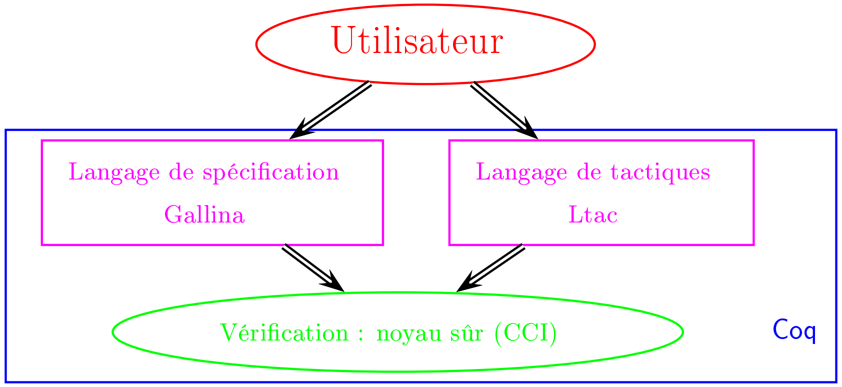
- signatures, définitions
- axiomes, énoncés

Preuves de propriétés

- construction interactive des preuves
- vérification déterministe

Certification de programmes

Architecture



Implantation

- Développé à l'INRIA en collaboration avec l'ENS Lyon puis l'université Paris Sud.
- Logiciel "libre" (LGPL) .
- 77 k lignes de code (Ocaml,Camlp4)
- 37 k lignes de bibliothèques
- 350 k lignes de développements dans les contributions.

Applications

Formalisations logiques π-calcul, Unity, CCS, CBS ...

Sémantique : ML, Java ...

Modélisations de protocoles : CSET, ABR, PIM, PGM ...

Théories mathématiques
Catégories, nombres réels, calcul formel, théorème fondamental de l'algèbre ...

Applications - programmes

Outils : Compilateur Esterel, Lustre, Coq en Coq

Procédures de décision : Algorithme de Buchberger, égalité dans les anneaux, arithmétique, tautologies, BDDs, model-checker, réécriture ...

Programmes impératifs Find, tris ...

Historique - Théorie

1985 Thèse de Th. Coquand introduisant le Calcul des Constructions.

1986 Ajout d'une hiérarchie d'univers.

1989 Ajout de définitions inductives primitives.

1994 Traitement des inductifs par point fixe gardé.

1995 Ajout de définitions co-inductives primitives.

Futur : 200?

- Modules primitifs
- Réécritures
- Suppression des types dans les termes
- Réduction optimisée des termes

Historique - Langage de haut niveau

1985 Mécanisme de sections

1986 Tactiques

1996 Définitions par filtrage
Arguments implicites
Coercions

2000 Langage de tactiques

Coq V7

- Environnement et noyau de vérification fonctionnels (J.-C. Filliâtre)
- Compilation de Gallina (H. Herbelin)
- Langage de tactiques (D. Delahaye)
- Extraction (J.-C. Filliâtre & P. Letouzey)
- Réduction (B. Barras)

Autres outils de modélisation et preuve

Proofs for Types and Programs

TYPES Groupe européen `http://www.durham.ac.uk/TYPES/`

- Assistants : Ecosse (LEGO, PLASTIC), Suède (ALF,ALFA,AGDA), Allemagne (MUTI).
- Modularité, sous-typage, certification de programme, preuves mathématiques.

Higher-Order Logic (HOL)

Logique classique, les fonctions vues comme des relations.

HOL Cambridge (Gordon), Intel (Harrison)
Preuves de matériel

Isabelle Cambridge (Paulson) & Munich (Nipkov)
"Framework" (HOL,ZF,. . .), unification d'ordre supérieur, protocoles cryptographiques.

PVS SRI (Owre, Rushby, Shankar)
Ajout de types concrets, sous-types.
Automatisation des preuves, abstraction.
Intégration Model-Cheking - preuves interactives.

Conférence TPHOLs
(20-23 août `http://shemesh.larc.nasa.gov/tphols2002/`)

Autres outils (B)

- Logique classique, théorie des ensembles (cadre non typé).
- Programmes annotés : machines abstraites.
- Méthodologie de développement (raffinement, implémentation).
- Outil de preuve externe.

`http://archive.comlab.ox.ac.uk/formal-methods/b.html`

References

[1] Eduardo Giménez. A tutorial on recursive types. Rapport Technique 0221, INRIA, 1998.

[2] G. Huet, G. Kahn, and Ch. Paulin-Mohring. *The* Coq *Proof Assistant - A tutorial - Version 7*, April 2001.

[3] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V7.2*, February 2002.

Le langage de spécification

Benjamin Werner

Les fonctions contre les ensembles

Principe 1 :

L'utilisateur de Coq veut tout, tout de suite et ensemble

Il veut pouvoir parler

- de nombres, de listes, d'arbres,
- du temps, de l'espace,
- de fonctions, d'ensembles, de relations, ...

Il a besoin de tout le langage mathématique

La langage mathématique permet de parler

- de fonctions

  $f =$

  $$\mathbb{N} \to \mathbb{N}$$

  $$x \mapsto (x \ mod \ 2)$$

- d'ensembles

  $$E = \{x \in \mathbb{N} \mid \exists y \in \mathbb{N} \ (x = 2 * y)\}$$

- de relations

  $$(R \ x \ y) \Leftrightarrow \exists z \in \mathbb{N} \ (x = z * y)$$

(reste les preuves. . .)

Redondance

Les choix de la théorie des ensembles :

$$R = \{(x, y) \in \mathbb{N} \times \mathbb{N} \mid \exists z \in \mathbb{N}\ (x = z * y)\}$$

$$f = \{(x, y) \in \mathbb{N} \times \mathbb{N} \mid y = (x\ mod\ 2)\}$$

Mauvais choix : $(5, 1) \in f$ mais pas $f(5) = 1$ ni $f(5)$

Il faut choisir un bon formalisme

Principe 2 :

Donner le premier rôle aux fonctions

Les ensembles et les relations sont exprimées comme des fonctions (à suivre)

L'opération de base pour former des objets n'est pas $\{x \mid P\}$ mais $x \mapsto t$ (on note $\lambda x\ t$ ou `[x]t` en Coq)

L'opération de base pour utiliser des objets n'est pas $x \in E$ mais $f(x)$, `(f x)` en Coq

$\boxed{\text{Les objets de Coq sont des programmes fonctionnels}}$

- Coq contient un langage de programmation fonctionnel
- C'est bien pour parler de programmes / d'informatique

La β-réduction

$$(([x](x\ mod\ 2))\ 5) = (5\ mod\ 2)$$

$$([x]t\ u) = t[x \leftarrow u]$$

Il y a une notion de calcul

autre avantage:

$\boxed{\text{On raisonne “gratuitement” modulo le calcul}}$

$$(([x](x\ mod\ 2))\ 5) = (5\ mod\ 2) = 1$$

Le domaine de définition d'une fonction

Une fonction a toujours un domaine de définition

$$\mathbb{N} \to \mathbb{N}$$

$$x \mapsto (x\ mod\ 2)$$

Principe 3 :

Il est *interdit* d'appliquer une fonction hors de son domaine

1/0 est prohibé par la *grammaire*

Principe 3 *bis* :

Il est interdit d'interdire. En appliquant une fonction hors de son domaine, on obtient un objet conventionnel (0, $\emptyset$, ...)

$1/0 = 0$ mais $x * (1/x) = 1$ n'est vrai que si $x \in \mathbb{R}^*$

Principe 3 *ter* :

En appliquant une fonction hors de son domaine, on obtient un objet inconnu (peut être 0, peut être 42, peut être π, ...)

$x * (1/x) = 1$ n'est vrai que si $x \in \mathbb{R}^*$

Dans les cas 3 *bis* et 3 *ter*

$$(u \in E) \Rightarrow (((E \rightarrow F/[x]t)\ u) = t[x \leftarrow u])$$

Peut-on s'affranchir de cette condition, i.e. étendre la définition de $(E \rightarrow F/[x]t)$ de manière à ce que

$$((E \rightarrow F/[x]t)\ u) = t[x \leftarrow u] \ ?$$

Lourd de conséquences : toute fonction a un point fixe

$$([x](f\ (x\ x)))([x](f\ (x\ x)))$$

neg : négation sur les booléens, $\mathcal{V}$: prédicat qui reconnait *true*

$$\mathcal{V}(neg(x)) \Leftrightarrow \neg\mathcal{V}(x)$$

Contradictoire

Choix :

- prédicat $\mathcal{V}$ gardé,
- β-réduction gardée,
- principe 3 (prohibition par la grammaire)

Prohibition par la grammaire

Des domaines aux types

- N'importe quel ensemble de définitions
- Prohibition de l'application d'une fonction hors de son domaine

Appartenance au langage indécidable

$$1/stop(p,n)$$

$stop(m,n) = 0$ si le programme p fini après n cycles, 1 sinon

grammaticale si p ne termine pas

Les types

Restriction à des ensembles décidables (i.e. tels que $t \in E$ soit décidable) : les types

Pour commencer:

- quelques types de base
- si T et U sont des types $T \rightarrow U$ est un type

$\mathbb{N}$ est un type `nat`

$\mathbb{N} \rightarrow \mathbb{N}$ est un type `nat -> nat`

$\mathbb{N}^*$ n'est pas un type

Tout objet a un type unique

Le type d'un objet indique seulement son degré de fonctionnalité (nombre, fonction, fonctionnelle, ...) mais pas davantage

Types en logique et en informatique

Les types ont été inventés par les logiciens (Russel, *Principia Mathematica*)

Ils sont surtout populaires chez les informaticiens.

Ce sont des ensembles définis par une unité de *structure*.

On peut appliquer les mêmes algorithmes aux objets d'un même type.

$\boxed{\text{Les objets de Coq sont des programmes fonctionnels typés}}$

C'est essentiellement le *noyau fonctionnel* du langage ML.

Fonctions à plusieurs arguments

Un point de notation:

il y a isomorphisme entre $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ et $\mathbb{N} \to (\mathbb{N} \to \mathbb{N})$

$$(x, y) \mapsto t \qquad \text{et} \qquad x \mapsto (y \mapsto t)$$

version *curryfiée* de la fonction.

On note $A \to B \to C$ pour $A \to (B \to C)$

On note $(f\ a\ b)$ pour $((f\ a)\ b)$

Pour être pédant:

- $\to$ associative à droite
- application associative à gauche

Les fonctions à plusieurs arguments (2)

$$+ : nat \to nat \to nat$$

$(+\ 5) : nat \to nat$ est la fonction qui ajoute 5 à son argument

Les types inductifs

Types de base (entiers, listes, arbres, ...) définis.

Les entiers définis abstraitement (pas des cardinaux)

- il existe un objet 0 qui est un entier et un objet S qui est une fonction des entiers dans les entiers (constructeurs)
- $S(x) = S(y) \Rightarrow x = y$ (les constructeurs sont injectifs)
- $\neg 0 = S(x)$ (non confusion des constructeurs)
- Récurrence (pas d'autres objets du type que ceux engendrés par les construteurs)

Définition inductive des entiers

Pour résumer:

Le plus petit type clos par deux constructeurs 0 et S distincts

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

constante particulière `Set`

correspond à Caml :

```
type nat = O | S of nat;;
```

définition par filtrage:

```
Definition pred := [n:nat]
  Cases n of O => O
          | (S p) => p
   end.
```


Récurrence structurelle

Définition “habituelle” de l'addition:

```
Fixpoint plus [n:nat] : nat -> nat :=
 [m]Cases n of
    O => m  | (S p) => (S (plus p m))
end.
```

Cette fonction termine car l'argument `n` décroit *structurellement*.

alternative:

```
Fixpoint plus [n ,m :nat] : nat :=
 Cases m of
    O => n  | (S p) => (S (plus n p))
end.
```


Autres définitions inductives

booléens, mais aussi:

```
Inductive Set List := nil:List
                    | cons:nat -> List -> List.

Inductive Set Tree := leaf:Tree
                    | node:Tree -> Tree -> Tree.


Inductive Set Couples := couple:nat -> nat -> Couples.
```


La β-réduction (suite)

Le filtrage et la recurrence (structurelle) étendent la β-réduction.

Sur les termes typés, la β-réduction termine, tout terme a une forme normale unique

Un entier est de la forme `O` ou successeur

On identifie les propositions équivalentes

Pas d'axiome de β-réduction

Les ensembles et les relations

L'ensemble

$$E = \{x \in \mathbb{N} \mid \exists y \in \mathbb{N}\ (x = 2 * y)\}$$

est la fonction qui à x associe la proposition $x \in E$

Il a donc le type $nat \rightarrow Prop$

$x \in E$ s'écrit donc $(E\ x)$

Une relation binaire sur les entiers est donc de type

$$nat \rightarrow nat \rightarrow Prop$$

La constante spéciale `Prop` est le type des propositions

Connecteurs et quantificateurs

"et", "ou" sont des objets de type

```
and : Prop -> Prop -> Prop
or : Prop -> Prop -> Prop
```

noté `A/\B` et `A\/B`.

implication notée `A->B`

$\forall x : A.P$ noté `(x:A)P`

$\exists x : A.P$ noté `(EX x:A|P)`

Les démonstrations objets de la théorie

- les entiers sont des objets du discours,
- les fonctions sont des objets du discours,
- les ensembles sont des objets du discours,
- les propositions sont des objets du discours,
- les démonstrations sont des objets du discours.

Principe de Brouwer-Heyting-Kolmogorov

Une démonstration de $A \Rightarrow B$ est une fonction qui associe une démonstration de B à toute démonstration de A.

S'utilise de la même manière, se fabrique de la même manière

Fonction de type $pr(A) \rightarrow pr(B)$

$$pr(A \Rightarrow B) = pr(A) \rightarrow pr(B)$$

Isomorphisme (Curry-De Bruijn-Howard) entre types et propositions

Notation `->` pour la flèche des types et pour l'implication

$$[x : A][y : B]x$$

démonstration de $A \Rightarrow B \Rightarrow A$

Une démonstration de $\forall x : T\ (P\ x)$ est une fonction qui associe une démonstration de $(P\ t)$ à tout objet de type T

Fonction de type `(x:T)(P x)`

Notation `(x:T)(P x)` pour le "quelque soit".

Une démonstration de $A \wedge B$ est un couple composé d'une démonstration de A et d'une démonstration de B.

Type des couples, type inductif, noté `/\`.

Idem pour `\/`, `~`, `Ex`, ...

Types paramétrés

Type des tableaux de longueur n et d'objets de type T

$$List : Set \qquad Tab : nat \to Set \to Set$$

$$(Tab\ 5\ Tree) : Set$$

Fonction qui à n associe tableau de longueur n

$$nat \to (Tab\ n\ Tree)$$

$$\Pi n : nat\ (Tab\ n\ Tree)$$

```
(n:nat)(Tab n Tree)
```


Les fonctions dont le domaine n'est pas un type en entier

$x \mapsto 1/x$

- Fonction de type `R -> R` étendue par une valeur quelconque en 0.
- Fonction de type `(x:R)((~(x = 0)) -> R)`

Quelques procédés de définition d'ensembles

En compréhension

$$\{x : \mathbb{N} \mid \exists y : \mathbb{N}\ (x = 2 * y)\}$$

```
[x:nat](EX y:nat | (x = (mult (S (S O))  y)))
```

Définition inductives

E défini par

- $0 \in E$,
- $x \in E \Rightarrow x + 2 \in E$

Accessibilité dans un graphe, grammaire, ...

Pas de création d'un nouveau type (les entiers pairs sont des entiers)

Exprimable par compréhension

```
[n:nat](P:nat -> Prop)
          (P O) ->
              ((x:nat)(P x) -> (P (S (S x)))) -> (P n)
```


Autre méthode

Création d'un nouveau type inductif des démonstrations de (Ex)

```
Inductive E:nat -> Prop :=
  c1:(E O)
| c2:(x:nat)(E x) -> (E (S (S x))).
```

```
(c2 (S (S O)) (c2 O c1)):(E (S (S (S (S O)))))
```

Principe de récurrence associé à la définition : récurrence sur la structure de la démonstration de $(E\ n)$

Quelques procédés de définition de fonction

Définition explicite

`[x:nat](plus x x)`

Définition implicite (opérateur de descriptions)

"à x on associe le plus grand entier y tel que $y * y \leq x$"

"à x on associe le y tel $P(x, y)$"

nécessité de démontrer l'existence (voire l'unicité de y)

π démonstration de `(x:nat)(EX y:nat | (P x y))`

$(\pi\ n)$ démonstration de `(EX y:nat | (P n y))`

$fst(\pi\ n)$ témoin de l'existence

$[n : nat]fst(\pi\ n)$ fonction recherchée

Les différents items de Gallina

- définitions de fonctions (explicites, implicites, par cas, par récurrence, ...)
- définitions d'ensembles (types inductifs, prédicats inductifs, en compréhension, ...)
- démonstrations de théorèmes

En théorie, ce sont des définitions.

`Definition p := [x:A]x : (A -> A).`

Démonstration de $A \Rightarrow A$.

Tout un appareil d'aide à la preuve.

```
Theorem th:A -> A.
Intro.
Assumption.
Save.

Theorem th:A -> A.
Auto.
Save.
```

Tactiques

Quelques commandes

- Calculer: `Eval Compute in ...`
- Afficher une constante: `Print`
- Vérifier le type: `Check`
- Chercher un lemme: `Search`
- Lire un fichier: `Load`
- Charger un developpement: `Require`
- Effacer des définitions: `Reset`

Possibilité d'ajout de syntaxe: `(or A B)`↔`A\/B`

`(eq A a b)`↔`a=b`

`(S (S (S (S O))))` ↔`(4)` après `Require Arith.`

Le langage de preuves

Benjamin Werner

Objectifs

- Initiation aux tactiques de base
- "Apprendre à marcher"
- Bachotage + jouer (un langage à apprendre)
- Comprendre le lien avec le formalisme
- tactiques,
- termes de preuve,
- tactiques de base,
- tactiques automatiques,
- combiner les tactiques.

La construction de la preuve

1. On commence par énoncer la proposition (ou lemme, théorème).
2. On construit ensuite la preuve de manière *interactive*.
3. Comme la proposition est un type, on doit construire un *habitant* de ce type.
4. On le fait donc par petit bouts, *top-down*
   (on part de la fin).

Pour commencer: Calcul Propositionnel

Tout à la main pour l'instant.

Pas de panique !

Deux tactiques de base

```
Coq < Parameter A,B,C:Prop.
```

```
Coq < Lemma exemp1 : ((A->B)->C)->B->C.
1 subgoal

  ============================
   ((A->B)->C)->B->C
```

On a un seul *but courant*

Intros

`Intros` "pousse" les hypothèses dans le contexte:

```
Coq < Intros c b.
1 subgoal

  c : (A->B)->C
  b : B
  ============================
   C
```

Du point de vue de la construction de la preuve: c'est une λ-abstraction.

avant : `?1 : ((A->B)->C)->B->C`

après : `[c:(A->B)->C)][b:B]?2` avec `?2:C`.

Apply

La tatique `Apply` correspond à l'application:

```
Coq < Apply c.
1 subgoal

  c : (A->B)->C
  b : B
  ============================
   A->B
```

On est passé au terme de preuve partiel `[c:(A->B)->C][b:B](c ?3)` on finit par. . .


```
Coq < Intros a.
1 subgoal

  c : (A->B)->C
  b : B
  a : A
  ============================
   B

Coq < Exact b.
Subtree proved!
```


La commande `Qed` (ou `Save`) ajoute le terme crée à l'environnement.

```
Coq < Qed.
Intros c b.
Apply c.
Intros a.
Exact b.
exemp1 is defined

Coq < Print exemp1.
exemp1 = [c:((A->B)->C); b:B](c [_:A]b)
     : ((A->B)->C)->B->C
```


La tactique la plus simple

`Exact` permet de donner directement le terme preuve.

```
Coq < Lemma exemp1 : ((A->B)->C)->B->C.

Coq < Exact [c][b](c [a]b).
Subtree proved!

Coq < Qed.
Exact [c:?; b:?](c [a:?]b).
exemp1 is defined

Coq < Print exemp1.
exemp1 = [c:((A->B)->C); b:B](c [_:A]b)
     : ((A->B)->C)->B->C
```

Le système a inféré les types des abstractions.

```
Coq < Lemma tauto1 : (A->B->C)->A->(A->B)->C.
1 subgoal

  ============================
   (A->B->C)->A->(A->B)->C

Coq < Intros c a b.
1 subgoal

  c : A->B->C
  a : A
  b : A->B
  ============================
   C
```


L'implication est associative à droite.

```
Coq < Apply c.
2 subgoals

  c : A->B->C
  a : A
  b : A->B
  ============================
   A
subgoal 2 is:
 B
```

On a deux sous-buts


```
Coq < Exact a.
1 subgoal

  c : A->B->C
  a : A
  b : A->B
  ============================
   B

Coq < Apply b; Exact a.
Subtree proved!

Coq < Qed.
Intros c a b.
Apply c.
Exact a.
Apply b; Exact a.
tauto1 is defined
```


Disjonction

Prouver: `Left` passe de `A\/B` à `A`

`Right` passe de `A\/B` à `b`

Utiliser: `Elim`

```
1 subgoal

  ============================
   A\/B->A
```

```
Coq < Intros c; Elim c.
2 subgoals

  c : A\/B
  ============================
   A->A
subgoal 2 is:
 B->A
```


Conjonction

Prouver:

`Split` passe de `A/\B` à deux buts `A` et `B`

Utiliser: `Elim`


```
1 subgoal

  ============================
   A/\B->B/\A

Coq < Intros c ; Elim c.
1 subgoal

  c : A/\B
  ============================
   A->B->B/\A
```


Assumption

Cette tactique résoud le but si il apparaît dans les hypothèses.

```
1 subgoal

  a : A
  b : B
  ============================
   A

Coq < Assumption.
Subtree proved!
```

Sinon elle *échoue*.

```
1 subgoal

  a : A
  b : B
  ============================
   C

Coq < Assumption.
Error: No such assumption
```

Plus généralement, une tactique peut:

- résoudre le but,
- ou bien engendrer d'autres buts,
- ou bien échouer.

Tautologies en général

La tactique `Tauto` résoud tous les buts qui ne demandent que des manipulations sur les connecteurs logiques (tautologies intuitionistes).

```
1 subgoal

  D : Prop
  ============================
   ((A->B->C)->D)->A/\C->D

Coq < Tauto.
Subtree proved!
```

`Tauto` résoud le but ou échoue. `Intuition` tente de simplifier le but s'il ne peut être complètement résolu.

Combinateurs de tactiques

Le ';' permet d'appliquer une tactique *à tous les sous-buts* engendrés par la tactique précédente.

Pour appliquer une séquence de tactiques différentes à chaque sous-but, on utilise la syntaxe de la forme:

`Apply H ; [ Exact a | Assumption | Auto ].`

Tactiques de contrôle

- La tactique `Fail` échoue toujours.
- Si `tac` échoue, alors `Try tac` ne fait rien (rend le même but).
- `tac1 Orelse tac2` applique `tac1` puis `tac2` *si la première échoue*.
- `First [tac1 | ... | tacn ]` applique la première tactique qui n'échoue pas.
- `Solve [tac1 | ... | tacn ]` applique la première tactique qui résoud le but.
- `Idtac` ne fait rien.

Exemples : `Try (tac ; Fail)` résoud le but ou ne fait rien ; `(Intuition ; Fail)` est équivalent à `Tauto`.

Les tactiques `Auto` et `Trivial`

`Auto` effectue d'elle-même un certain nombre d'opérations:

- simplification par calcul (`Simpl`)
- `Intros`
- `Assumption`
- `Apply` des hypothèses locales
- surtout `Apply` avec les éléments d'une base de lemmes.
- puis réitération

On rajoute le lemme `T` dans la base par `Hints Resolve T.`

`Trivial` est similaire mais réitère moins (plus rapide)

`Auto` et `Trivial` n'échouent jamais, ni ne simplifient.

Soit elles resolvent le but, soit elles ne font rien.

`Auto` $\simeq$ `(Try Auto)` $\simeq$ `Try (Auto ; Fail)`

Calculer pour raisonner: `Simpl`

`Simpl` demande au système de calculer (simplifier) autant que possible.

```
Coq < Lemma ex_simpl : (plus (2)(2))=(4).
1 subgoal

  ============================
   (plus (2) (2))=(4)

Coq < Simpl.
1 subgoal

  ============================
   (4)=(4)
```


Analyse de cas: `Elim`

```
1 subgoal

  ============================
   (b:bool)(P b)

Coq < Intros b ; Elim b.
2 subgoals

  b : bool
  ============================
   (P true)
subgoal 2 is:
 (P false)
```

Alternative : `Destruct`

`Destruct d` $\simeq$ `Intro d ; Elim d`

Raisonnement par récurrence: `Elim`

Quand le type éliminé est récursif:

```
1 subgoal

  n : nat
  ============================
   (P n)
```

```
Coq < Elim n.
2 subgoals

  n : nat
  ============================
   (P O)
subgoal 2 is:
 (n0:nat)(P n0)->(P (S n0))
```


Induction

Alternative: sur le but `(n:nat)(P n)` il est équivalent de faire:

`Induction n.`

ou bien

`Intro n ; Elim n.`

La relation d'égalite (1)

On note l'égalité `a=b` si `a` et `b` sont du *même type* `A`.

C'est en fait une abréviation pour `(eq A a b)`.

Prouver: `a=a` se prouve par `Trivial`, `Reflexivity`, `Auto`...

Utiliser:

`Replace a with b` remplace `a` par `b` dans le but et demande à prouver `a=b`.

Egalité (2) `Rewrite`

On peut aussi fournir la preuve d'égalité:

```
1 subgoal

  e : x=y
  ============================
   (P x)
```

```
Coq < Rewrite -> e.
1 subgoal

  e : x=y
  ============================
   (P y)
```


Proprietés de l'addition

```
Fixpoint plus [n:nat] : nat -> nat :=
 [m]Cases n of O => m
           | (S p) => (S (plus p m))
  end.
```

Donc

`(plus O m)` se simplifie en `m`

`(plus (S n) m)` se simplifie en `(S (plus n m))`

Mais comment monter que `m=(plus m O)` ?

...par récurrence. Il faut raisonner !

Outils

Christine Paulin

Contenu

- Lancement de Coq
- Version native ou byte-code
- Documentation de Coq
- Bibliothèque standard et contributions des utilisateurs
- Explorer la bibliothèque standard et les contributions
- Organiser et documenter ses développements

Lancement de Coq

Mode textuel

- Sous Unix : commande `coqtop`
- Sous Windows (Mac) : ouvrir `coqtop`

Interaction

- L'invite : `Coq <`
- Commandes terminées par un `.`
  Exemple : `Pwd.`
- Commentaires `(* *)`
- Sortie : `Quit.`

Interfaces pour Coq

Éditeur de ligne : `ledit coqtop`
`http://cristal.inria.fr/~ddr`

Mode Emacs : Proof General
`http://zermelo.dcs.ed.ac.uk/~proofgen/`

Mode Graphique

- CtCoq (sous Unix, utilise Centaur)
  `http://www-sop.inria.fr/croap/ctcoq/ctcoq-fra.html`
- Pcoq (sous Unix, technologie Java)
  `http://www-sop.inria.fr/lemme/pcoq`

Code natif ou byte-code

`coqtop -byte` version byte-code,
chargement dynamique de code ocaml,
accès à la boucle d'interaction de ocaml,
version lente.

`coqtop` version en code natif, rapide.

La documentation de Coq

`http://coq.inria.fr/doc-fra.html`

"Reference Manual" `http://coq.inria.fr/doc/main.html`

"Addendum to the Reference Manual"

"Tutorial" de base

"Tutorial" sur les types récursifs.

Format des fichiers Coq

Sources
Gallina/Vernaculaire : suffixe `.v`

Format compilé
Termes typés (certifiés) : suffixe `.vo`

Chargement code source : `Load` *nom_du_fichier*

Chargement code compilé : `Require` *nom_du_fichier*

Compilation : `coqc` *nom_du_fichier*

La bibliothèque standard de Coq

Version 7 : `http://coq.inria.fr/library/`

Init	Chargé par défaut
Arith	Arithmétique de base
Zarith	Entiers binaires
Reals	Axiomatisation des réels
Bool	Booléens

La bibliothèque standard de Coq (suite)

Lists	Listes finies et infinies
IntMap	Structure efficace pour des ensembles finis
Logic	Logique classique et égalité dépendante
Sets	Définitions et résultats de base sur les ensembles
Relations	Définitions et résultats de base sur les relations
Wellfounded	Relations bien fondées

Contributions

Développements indépendants hébergés et maintenus sur notre site.
`http://coq.inria.fr/contribs-fra.html`

Recensement des développements:
`http://coq.inria.fr/results-fra.html`
`http://coq.inria.fr/sond-eng.html`

Contacts:
`http://coq.inria.fr/contacts-fra.html`
Coq-Club `coq-club@pauillac.inria.fr`
Bugs `coq-bugs@pauillac.inria.fr`

Documentation de la bibliothèque standard

- Document “The Standard Library”
- Répertoire `theories` de la distribution
- Recherche dans la bibliothèque standard
  - Commandes vernaculaire
    `Search` *ident*, `SearchPattern/SearchRewrite` *pattern*

Information sur un terme

Le type d'un terme `Check` *term*
Exemple : `Check nat`

La valeur d'un terme `Print` *term*
Exemple: `Print nat`

Recherche avec `Search`

Principe `Search` *ident* trouve les termes dans l'environnement courant dont la conclusion commence par *ident*.

Mode d'emploi Commande Coq : `Search` *ident*

Exemples Rechercher les termes qui peuvent former un objet dans `nat`.

`Search nat`.

Rechercher les théorèmes dont la conclusion porte sur `le`.

`Search le`.

Recommencer après avoir chargé la bibliothèque `Le`.

Recherche avec `SearchPattern`,`SearchRewrite`

Principe `SearchPattern` *motif* trouve les termes de la bibliothèque dont la conclusion filtre *motif*.

Mode d'emploi Commande Coq : `SearchPattern` *motif*

Exemple Rechercher les théorèmes qui portent sur la négation de `le`:
`SearchPattern ~(le ?  ?).`

Principe `SearchRewrite` *motif* équivalent de `SearchPattern` *motif*`=?` et `SearchPattern ?=`*motif*

Organiser un développement

Génération de Makefile

Création d'un fichier `Make` comportant les fichiers `.v`, `.ml` et sous-repertoires à compiler.

`coq_makefile -f Make -o Makefile`
`touch .depend`
`make depend` (commande `coqdep`)
`make opt` ou `make OPT=-opt` *`file.vo`* (commande `coqc`)

En savoir plus cf pages "man" ou le Manuel de référence

Documenter un développement

- `gallina`
  Le développement sans les tactiques.
- `coq-tex`
  Insère des commandes Coq dans un document LaTeX.
- `coqdoc`
  Produit un fichier LaTeX ou HTML dans lequel les commentaires sont interprétés
- génération XML

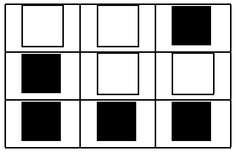
En savoir plus cf pages "man" ou le Manuel de référence

Le damier de Cachan

C. Paulin & B. Werner

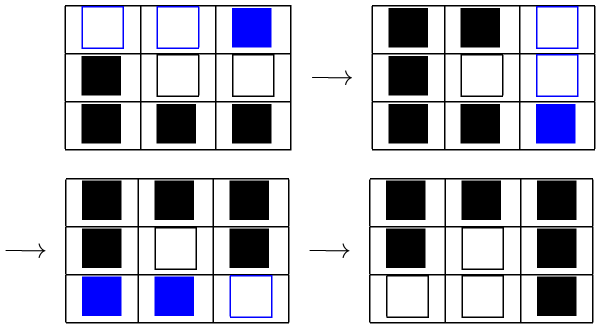
Le problème

On dispose d'un damier de 3 cases sur 3, sur lequel sont posés 9 jetons bicolores (une face blanche et une face noire).



À chaque étape on peut soit retourner les pions d'une ligne, soit retourner les pions d'une colonne.

Le problème



On cherche à savoir si certaines configurations sont ou non atteignables à partir d'une configuration initiale.

Modélisation des pions

En s'inspirant du type `bool`, définir un type `color` ayant exactement deux valeurs `White` et `Black`

Définir par cas la fonction `turn_color` de type `color→color` qui inverse les couleurs.

Tester la fonction définie :
`Eval Compute in (turn_color White)`

Triplets polymorphes

La modélisation du damier utilise un type polymorphe de triplets.

Ouvrir une section (`Section` *`nom_de_la_section`*) dans laquelle on introduit une variable `M:Set`.

Définir le type `triple` des triplets d'objets de type `M` avec un constructeur : `Triple` de type `M → M → M → triple`.

La fonction `triple_x` qui à `x:M` associe le triplet formé de trois `x` s'écrit :
`Definition triple_x:M→triple:=[x](Triple x x x).`

Fonctions sur les triplets

Pour accéder aux composantes du triplet on définit un type :
`Inductive pos : Set := A : pos | B : pos | C : pos.`

Définir `proj_triple:pos → triple → M` qui renvoie la valeur du triplet en la position donnée.

On se donne maintenant une fonction `f` de type M→M.

Définir `triple_map:triple → triple` qui applique la fonction `f` à chaque composante du triplet.

Définir `triple_map_select:pos → triple → triple` qui applique la fonction `f` à la composante désignée du triplet.

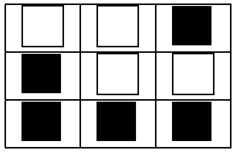
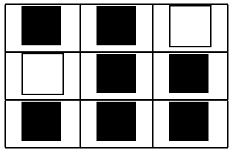
Fermer la section sur les triplets (`End` *`nom_de_la_section`*)
Observer le type des objets définis.

Modéliser le damier

Définir le type `damier` comme étant un triplet de triplet d'objets de type `color`.

Définir `white_board` le damier dont tous les pions sont blancs.

On introduit deux configurations :

`start` =  `target` = 

Fonctions sur les damiers

Définir `proj_board:pos → pos → board → color` la fonction qui calcule la couleur d'un point du damier.

Définir `turn_line`, `turn_col:pos → board → board` qui retournent respectivement une ligne et une colonne du damier.

Tester ces fonctions sur des exemples.

Déplacement

Définir une relation `move:board → board → Prop` telle que :
`(move b1 b2)` est vrai lorsque `b2` est obtenu en retournant une ligne ou une colonne de `b1`.

On pourra choisir une définition inductive ou bien utiliser les connecteurs logiques.

Dans les deux cas on définira :
`move_line : (b1:board)(p:pos)(move b1 (turn_line p b1))`
`move_col : (b1:board)(p:pos)(move b1 (turn_col p b1))`.

Déplacements multiples

Définir une relation `moves:board → board → Prop` telle que :
`(moves b1 b2)` est vrai lorsque `b2` est obtenu par 0, 1 ou plusieurs déplacements à partir de `b1`.

On prouvera les lemmes suivants :
`moves_init : (b1:board)(moves b1 b1)`
`move_moves : (b1,b2:board)(move b1 b2) → (moves b1 b2)`
`moves_trans: (b1,b2,b3:board)`
`(moves b1 b2)→(moves b2 b3)→(moves b1 b3)`

Résultats

Établir `(moves start target)` (facile)
et `~(moves start white_board)` (plus dur).

Pour établir que `~(moves b1 b2)` on montrera qu'il suffit de trouver un type `M` et une fonction `f:board→M` telle que :

`(b1,b2:board)(moves b1 b2)→(f b1)=(f b2)`
et `(f b1)` n'est pas égal à `(f b2)`.