Adapter Coq à ses besoins

Bruno Barras

Contenu

- Introduire ses propres notations (infixes, grammaire)
- Paramétrisation de tactiques (Auto, AutoRewrite)
- Constructions de tactiques
  (Tactic Definition, Meta Definition, tactique en ML)
- Comment faire une tactique autonome en utilisant une table.

Syntactic Definition

- Même syntaxe qu'une définition
- Typée uniquement à l'utilisation dans une définition non syntaxique
- Liaison statique des identificateurs

Exemple:

```
Coq < Syntactic Definition Eq := (eq ?).
Eq is now a syntax macro
Coq < Check (Eq O O).
O=O
     : Prop
Coq < Check [eq:nat](Eq O eq).
[eq0:nat]O=eq0
     : nat->Prop
```

Rq: pas de règle d'affichage associée.

Infixes simples

```
Coq < Implicit Arguments On.
Coq < Inductive prod' [A:Set; B:Set] : Set :=
Coq <   pair' : A->B->(prod' A B).
Coq < Infix 3 "**" prod'.
```

Infix introduit une règle de grammaire et une règle d'affichage:

```
Coq < Check bool ** nat.
bool**nat
     : Set
Coq < Infix 3 ",," pair'.
Coq < Check (O,,true).
O,,true
     : nat**bool
```

Niveau de priorité des opérateurs

- 11 niveaux de priorité prédéfinis (`constr0`, `constr1`, ...)
- le plus faible lie le plus fort

Niveau	Assoc	Constructeur
10	?	application, `EX`, `ALL`, `let`, `if`
9	R	`::`, `(_)_`, `[_]_`
8	R	`->`, `<->`
7	R	`\/`
6	R	`/\`
4	L	`+`
3	R	`*`
2	R	`~`
1	N	`Cases`, `=`, `==`, `{_|_}`
0	?	`Fix`, `CoFix`, atomes

Les précédences

```
Coq < Check (bool ** nat) -> nat.
bool**nat->nat
     : Set

Coq < Check (bool ** nat) == (nat ** bool).
(bool**nat)==(nat**bool)
     : Prop
```


Associativité

- Associativité à gauche par défaut
  (sinon utiliser LEFTA, RIGHTA, NONA: `Infix NONA 3 ...`)
- Seules les associativités des niveaux 6 à 9 peuvent être modifiées

```
Coq < Check bool ** (bool ** bool).
bool**(bool**bool) : Set

Coq < Check (bool ** bool) ** bool.
bool**bool**bool : Set
```


Grammaires générales: exemple

Exemple : une grammaire d'expressions

```
Grammar constr expression :=
  var [ variable($v) ] -> [ (Var $v) ]
| cte [ nat:number($n) ] -> [ (Cte $n) ]
| neg [ "neg" expression($e) ]
    -> [ (Fun f_minus (Cte (1)) $e) ]

with variable := ...    .
```

Grammaires générales

- Caractérisées par un nom de grammaire (ici `constr`) et des entrées (des symboles non-terminaux ; ici `expression` et `variable`)
- Doit être LL(1) et présentée factorisée au 1er niveau (Camlp4)
- 3 grammaires de base : `vernac`, `tactic` et `constr`
- on peut ajouter des clauses à un non-terminal existant, ajouter un non-terminal à une grammaire existante et créer une nouvelle grammaire
- les méta-variables liant les productions des non-terminaux commencent par un "$"

Les productions des grammaires

- Le type des productions de grammaire dépend de la grammaire étendue (terme, tactique ou commande vernac)
- Opérateurs: `let` et `case`
  Ex:

```
 let $x = [ (plus $c1 $c2) ] in (mult $x $x)
case [ $c ] of
  ($x:$a)$b -> [ $c1 ]
| $_ -> [ $c2 ]
esac
```


Les règles d'affichage : exemple

```
Syntax constr
  level 3:
    f_conj [ (Fun f_conj $e1 $e2) ]
     -> [[<hov 0> $e1:L [0 0] "/\\" $e2:E ]];
  level 2:
    f_neg [ (Fun f_minus (Cte (1)) $n) ] -> [ "neg " $n:E ];
  level 1:
    f_eq [ (Fun f_eq $e1 $e2) ]
     -> [ [<hov 0> $e1:L [0 0] "=" $e2:L ] ];
  level 0:
    Var [ (Var $n) ] -> [ $n ]
  | cte [ (Cte $n) ] -> [ $n ].
```


Les règles d'affichage

- classées par grammaire (constr, tactic, vernac)
- définies par filtrage sur le terme à afficher
- classées par niveau de précédence (avec affichage automatique de parenthèses) grâce aux `:E`, `:L`
- indentation: empilement de boîtes `[<...> ...]`, et d'espaces sécables `[`m n`]`

Exercice 1

Écrire les commandes Grammar et Syntaxe équivalentes aux définitions infixes de l’exemple:

```
Infix 3 "**" prod'.
Infix 3 ",," pair'.
```

Attention: les arguments implicites apparaissent dans le terme filtré par les règles d’affichage.

Exercice 1 (solution)

```
Coq < Grammar constr constr3 :=
Coq <   prod' [ constr3($l) "**" constr2($r) ] -> [(prod' $l $r)]
Coq < | pair' [ constr3($l) ",," constr2($r) ] -> [(pair' $l $r)].
<W> Grammar extension: some rule has been masked
<W> Grammar extension: some rule has been masked

Coq < Syntax constr level 3:
Coq <   prod' [(prod' $A $B)] -> [ [<hov 0> $A:E "**" [0 0] $B:L ] ]
Coq < | pair' [(pair $_ $_ $x $y)] ->
Coq <          [ [<hov 0> $x:E ",," [0 0] $y:L ] ].
```


Exercice 2: affichage de permutations

Écrire les commandes permettant d’afficher les étapes successives d’une permutation.
Ex: p : `(permut [1;2;3;4] [1;4;3;2])` affiché

```
 1  *2   3  4
 1  *3   4  2
 1   4  *2  3
 1   4   3  2
```


Exercice 2 (suite)

```
Coq < Section Permut.

Coq < Variable A : Set.

Coq < Inductive permut: (list A) -> (list A) -> Prop :=
Coq <   Prefl: (l:(list A)) (permut l l)
Coq < | Ptail: (n:A)(l1,l2:(llist A)
Coq <            (permut l1 l2) ->
Coq <            (permut (cons x l1) (cons x l2))
Coq < | Prota: (n:A)(l1,l2:(list A))
Coq <            (permut (app l1 (cons x (nil ?))) l2) ->
Coq <            (permut (cons x l1) l2).

Coq < End Permut.
```

```
Syntax constr level 0:
    RL_nil  [(RLIST (nil $_) $1)] -> [<<(LIST $1)>>]
  | RL_cons [(RLIST (cons $x $10) $1)] -> [<<(RLIST $10 (cons $x $1))>>]
  | L_cons    [(LIST (cons $x $1))] -> [ $x "  " <<(LIST $1)>> ]
  | L_nil     [(LIST (nil $_))] -> [ ]
  | L_one     [(LIST (cons $x (nil $_)))] -> [ $x ]

  | Pfin [(PERMUT $1 (Prefl $_ $10))] ->
         [ <<(RLIST $1 (nil ?))>> "  " <<(LIST $10)>>]
  | Ptail [(PERMUT $1 (Ptail $_ $a $_ $_ $p))] ->
          [<<(PERMUT (cons $a $1) $p)>>]
  | Prota [(PERMUT $1 (Prota $A $x $11 $12 $p))] ->
      [ <<(RLIST $1 (nil ?))>> " *" $x "  " <<(LIST $11)>> FNL
        <<(PERMUT $1 $p)>> ]
  | Prefl0 [(Prefl $_ $1)] -> [<<(LIST $1)>>]
  | Ptail0 [(Ptail $A $a $10 $11 $p)] ->
           [<<(PERMUT (nil ?) (Ptail $A $a $10 $11 $p))>>]
  | Prota0 [(Prota $A $a $11 $12 $p)] ->
           [<<(PERMUT (nil ?) (Prota $A $a $11 $12 $p))>>].
```


Automatisation: bases d'aides pour `Auto` ("Hints")

- Chaque module de la bibliothèque standard a sa propre base d'aide (`core`, `arith`, `sets`, `datatypes`,...)
- Aides classés par constante de tête et pondéré
- Appliqués par `Auto` selon la constante de tête du but à résoudre

Pourquoi séparer en bases ?

- Efficacité: une base contenant à la fois $A \wedge B \Rightarrow A$ et $A \Rightarrow B \Rightarrow A \wedge B$ serait *très* inefficace

Exemple d'aides pour `Auto` ("Hints")

- `Hints Resolve in_eq : list.`
- `Hints Immediate eq_add_S : core.`
- `Hints Unfold lt : core.`
- `Hint le_Omega:zarith:=Extern 10 (le ? ?) Abstract Omega.`

Utilisation:

- `Auto.`
- `Auto with sets arith.`
- `Auto with *.`

Automatiser la réécriture: `AutoRewrite`

`AutoRewrite [` $base_1 \ldots base_n$ `] using` *tac*

- Réalise toutes les réécritures selon $base_1 \ldots base_n$
- Utilise *tac* pour le sous-but principal

`Hint Rewrite -> [` $term_1 \ldots term_n$ `] in` *ident* `using` *tac*
`Hint Rewrite <- [` $term_1 \ldots term_n$ `] in` *ident* `using` *tac*

- Ajoute $term_1 \ldots term_n$ à *ident*
- Le sens peut être précisé
- Utilise *tac* pour les sous-buts, le sous-but principal exclu

Utiliser `AutoRewrite`: fonction de Mac Carthy

```
Require Omega.

Variable g:nat->nat->nat.

Axiom g0:(m:nat)(g (0) m)=m.
Axiom g1:
  (n,m:nat)(gt n (0))->(gt m (100))->(g n m)=(g (pred n) (minus m (10))).
Axiom g2:(n,m:nat)(gt n (0))->(le m (100))->(g n m)=(g (S n) (plus m (11))).

Hint Rewrite [ g0 g1 g2 ] in base using Omega.

Lemma Ret91:(g (1) (95))=(91).
Proof.
  AutoRewrite [ base ] using Reflexivity Orelse Simpl.
Save.
```


Tactiques

Bruno Barras

Notion de tactique

Définition:

- C'est une fonction qui prend un but et rend d'autres (sous-)buts
- Processus de raffinement
- Pas (élémentaires ou non) des règles du calcul

Mécanisme sous-jacent:

- Manipulation d'un arbre de preuve
- Production d'un terme preuve (Curry-Howard)

Construction de tactiques

Deux types de tactiques:

- Tactiques primitives (`Intro`, `Induction`, ...)
- Tactiques élaborées (`Tauto`, `Omega`, ...)

Écrire des tactiques élaborées:

- Combinateurs (*tacticals*):
  à toplevel (dans les preuves / `Tactic Definition`), limité
- Métalangage (Objective Caml):
  interfaçage moins évident, Turing-complet

Les tacticals

- `Idtac`, `Fail`
- `Do` n *tac*, `Repeat` *tac*
- tac_1`;`tac_2, tac_0`;[`tac_1`|`...`|`tac_n`]`
- tac_1 `Orelse` tac_2, `Try` *tac*
- `First [`tac_1`|`...`|`tac_n`]`, `Solve [`tac_1`|`...`|`tac_n`]`
- (`Info` *tac*, `Abstract` *tac*)

Métalangage et systèmes à la LCF

Système LCF:

- Logic for Computable Functions
- Robin Milner, à Edinburgh, début 70
- Métalangage (ML) pour écrire les tactiques
- Métalangage distinct du langage d'implantation (Stanford Lisp)

Évolution:

- Edinburgh LCF → Cambridge LCF (portage en Franz Lisp)
- À Cambridge: Cambridge LCF → HOL
- À l'INRIA: Cambridge LCF → Coq

Nouveau langage de tactiques ($\mathcal{L}_{tac}$) de la V7

Situation:

- Tacticals: faciles d'accès, portable, limité
- Métalangage: complet, complexe, peu portable

Motivations:

- Petites automatisations
- À *toplevel*
- Portable
- Contrôle de la syntaxe

⇒ Redonner au métalangage la place qu'il occupait

Définition de $\mathcal{L}_{tac}$

Intégré au toplevel; inclut les tacticals.

On fournit:

- Un petit noyau fonctionnel (`Fun`, `Let ... In`)
- Récurseurs (`Rec`)
- Filtrage (avec *backtrack*) sur les (sous-)termes (`Match`)
  sur les contextes (`Match Context`)

Peut être utilisé:

- Dans les scripts
- Dans des définitions à toplevel (`Tactic`/`Meta Definition`)

Un premier petit exemple: `nat`$\neq$ `bool`

$$\vdash (\exists x:\mathbb{N}.\exists y:\mathbb{N}.\forall z:\mathbb{N}.x=z \lor y=z) \to \bot$$

$$\boxed{\text{Skolémisation avec } a \text{ et } b}$$

$$\downarrow$$

$$\forall z:\mathbb{N}.a=z \lor b=z \vdash \bot$$

$$\boxed{\text{Élimination avec 0, 1 et 2}}$$

$$\downarrow$$

$$\begin{array}{c} \text{(8 cas)} \\ a=2, a=1, a=0 \vdash \bot \\ b=2, a=1, a=0 \vdash \bot \\ \vdots \\ b=2, b=1, b=0 \vdash \bot \end{array}$$

$$\boxed{\text{Transitivité}}$$

Preuve à la main

```
Lemma nat_bool0: ~(EX x:nat|(EX y:nat|(z:nat)(x=z)\/(y=z))).
Proof.
  Red;Intro H.
  Elim H;Intros a Ha.
  Elim Ha;Intros b Hb.
  Elim (Hb (0));Elim (Hb (1));Elim (Hb (2));Intros.
  Cut (0)=(1);[Discriminate|Apply trans_equal with a;Auto].
  Cut (0)=(1);[Discriminate|Apply trans_equal with a;Auto].
  Cut (0)=(2);[Discriminate|Apply trans_equal with a;Auto].
  Cut (1)=(2);[Discriminate|Apply trans_equal with b;Auto].
  Cut (1)=(2);[Discriminate|Apply trans_equal with a;Auto].
  Cut (0)=(2);[Discriminate|Apply trans_equal with b;Auto].
  Cut (0)=(1);[Discriminate|Apply trans_equal with b;Auto].
  Cut (0)=(1);[Discriminate|Apply trans_equal with b;Auto].
Save.
```


Preuve en utilisant $\mathcal{L}_{tac}$

```
Lemma nat_bool1: ~(EX x:nat|(EX y:nat|(z:nat)(x=z)\/(y=z))).
Proof.
  Red;Intro H.
  Elim H;Intros a Ha.
  Elim Ha;Intros b Hb.
  Elim (Hb (0));Elim (Hb (1));Elim (Hb (2));Intros;
  Match Context With
  | [_:?1=?2;_:?1=?3|-?] ->
    Cut ?2=?3 ;[Discriminate|Apply trans_equal with ?1;Auto].
Save.
```


Permutation sur des listes closes

```
Tactic Definition Permut n :=
  Match Context With
  | [|-(permut ? ?1 ?1)] -> Apply Prefl
  | [|-(permut ? (cons ?1 ?2) (cons ?1 ?3))] ->
    Let newn = Eval Compute in (length ?2) In
    Apply Ptail;Permut newn
  | [|-(permut ?1 (cons ?2 ?3) ?4)] ->
    (Match Eval Compute in n With
    | [(1)] -> Fail
    | _ -> Apply Prota;Compute;Permut '(pred n)).

Tactic Definition PermutProve :=
  Match Context With
  | [|-(permut ? ?1 ?2)] ->
    (Match Eval Compute in ((length ?1)=(length ?2)) With
    | [?1=?1] -> (Permut ?1)).

Lemma p1: (permut ? (cons (1) (cons (2) (cons (3) (cons (4) (nil ?)))))
                    (cons (1) (cons (4) (cons (3) (cons (2) (nil ?)))))).
PermutProve.
Save.
```

D'autres exemples

Possibilité de résoudre des problèmes non triviaux:

- Réimplantation de la tactique `Tauto`
- Égalités entre types et modulo isomorphismes
- Procédure de Davis-Putnam-Logemann-Loveland
- Tactique `Field`

De manière générale:

- À toplevel (dans les sources)
- Code court, compact et plus simple
- Gain en lisibilité $\Rightarrow$ maintenance accrue

Extensions

Ouverture vers Objective Caml:

- Quotation pour utiliser $\mathcal{L}_{tac}$ en Objective Caml
- Véritable API pour Coq
- Extensions de tactiques (`Tauto`)

Développement d'un *debugger* spécifique:

- Dédié aux tactiques
- Utile pour les tactiques qui peuvent backtracker

Exercice 1

Montrer qu'une **constante** (entière positive) non nulle de $\mathbb{R}$ est non nulle.

Indications et suggestions:

- Schéma: $c \neq 0 \Leftarrow\ c > 0 \Leftarrow\ c > c - 1$ et $c - 1 > 0$
- Constantes à utiliser: `R0`, `R1`, `Rplus`, `Rgt`
- Lemmes à utiliser:
  `R1_neq_R0`, `Rgt_not_eq`, `Rlt_R0_R1`,
  `Rgt_trans`, `Rplus_sym`, `Rlt_r_r_plus_R1`
- Pretty-print: “3“ $\equiv$ `(Rplus R1 (Rplus R1 R1))`

Exercice 2

Implanter l'algorithme de tri bulle sur des listes Coq.

Indication:

Voir l'exemple sur les permutations sur des listes closes.

Tactiques en ML

- Manipulation des termes comme type abstrait
- Mise à disposition des tactiques et tacticals (`tclIDTAC`, `tclTHEN`, `tclORELSE`,...)
- Possibilité d'avoir un état global (base de théorèmes, table,...)

Problème: synchronisation avec `Reset`, passage aux modules

Domaine de $\mathcal{L}_{tac}$

Choix entre ML et $\mathcal{L}_{tac}$:

Quotation

ML	$\mathcal{L}_{tac}$
Boîte noire	Dans les sources
Compilé	Interprété
Structures de données de ML	Structures de données de Coq
Manipulation de l'environnement	Pas de manipulation de l'environnement

Certification de programmes

J.-C. Filliâtre

Contenu

- Certification de programmes fonctionnels
  - Approche directe
  - Extraction (+ TD)
- Tactiques certifiées : technique de réflexion (+ TP)
- Certification de programmes impératifs (+ TD)

Certification de programmes fonctionnels

Approche directe : on définit une fonction `f` dans `Coq` et on établit des propriétés satisfaites par `f`

Exemple :

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

puis

```
Lemma plus_sym : (n,m:nat)(plus n m)=(plus m n).
Intros n m ; Elim n ; ...
```


Extraction

Une preuve `Coq` est constructive : on peut en extraire un programme

Le programme correspond à la partie calculatoire (dans `Set`) ;

les arguments logiques (dans `Prop`) sont supprimés

Le programme extrait est certifié correct

Exemple : L'extraction d'une preuve de

```
(n:nat)(even n) -> { p:nat | n=(plus p p) }
```

est une fonction calculant `p` étant donné `n` pair

Extraction dans le toplevel Coq

`Extraction` *term* `.`

Le programme extrait est donné en syntaxe Ocaml :

```
Coq < Extraction plus.
let rec plus n m =
  match n with
    O -> m
  | S p -> S (plus p m)
```


Extraction dans le toplevel, récursivement

`Recursive Extraction` $id_1 \ldots id_n$ `.`

```
Coq < Check even_2n.
even_2n
     : (n:nat)(even n)->{p:nat | n=(double p)}

Coq < Recursive Extraction even_2n.
type nat =
    O
  | S of nat

let rec div2 = function
  O -> O
| S n0 -> (match n0 with
             O -> O
           | S n' -> S (div2 n'))

let even_2n n =
  div2 n
```

Extraction dans un fichier

`Extraction "`*fichier*`"` $id_1 \ldots id_n$`.`

Extraction modulaire :

`Extraction Module` M`.`

`Recursive Extraction Module` M`.`

Programme Ocaml par défaut

Programme Haskell en ajoutant le préfixe `Haskell` à ces commandes

Options de l'extraction

Réalisation d'axiomes :

`Extract Constant` *id* `=> string.`

`Extract Inductive` *id* `=> string [ string .. string ].`

Optimisation du code extrait :

`Set/Unset Extraction Optimize.`

`Set/Unset Extraction AutoInline.`

`Extraction Inline` id_1 `..` id_n`.`

`Extraction NoInline` id_1 `..` id_n`.`

Méthodologie

L'extraction peut être utilisée simplement pour obtenir le code ML correspondant à des fonctions Coq purement calculatoires (dans un but de tests, d'animations, de combinaison avec du code existant, etc.)

Cependant, il est souvent plus rapide d'extraire la fonction d'une preuve de sa spécification.

Exemples

Égalité sur les entiers naturels

```
Fixpoint eq_nat [n,m:nat] : bool :=
  Cases n m of
  | O O => true
  | (S n') (S m') => (eq_nat n' m')
  | _ _ => false
  end.
```

puis

```
Lemma eq_nat_correct : (n,m:nat) (eq_nat n m)=true <-> n=m.
```

Exemples (suite)

On préfère montrer directement (dans `Peano_dec`)

```
Lemma eq_nat_dec : (n,m:nat){n=m}+{~n=m}.
Proof.
Induction n; ...
```

La fonction extraite a le type escompté

```
Coq < Extraction (n,m:nat){n=m}+{~n=m}.
nat -> nat -> sumbool

Coq < Extraction sumbool.
type sumbool = left | right
```


Exemples (suite)

La fonction extraite est la fonction escomptée :

```
Coq < Extraction eq_nat_dec.
let rec eq_nat_dec n m =
  match n with
    O -> (match m with
            O -> left
          | S n0 -> right)
  | S n0 -> (match m with
               O -> right
             | S n1 -> eq_nat_dec n0 n1)
```


De manière générale

Il peut être préférable de définir le type inductif adapté à la spécification plutôt que de combiner les types existants (`sumbool`, `exists`, `and`, ...)

Exemple : une liste d'entiers pairs

```
Inductive even_list : Set :=
  | Enil : even_list
  | Econs : (n:nat)(even n) -> even_list -> even_list.
```

```
Coq < Extraction even_list.
type even_list =
    Enil
  | Econs of nat * even_list
```


Tactiques certifiées : technique de réflexion

Tactique traditionnelle = morceau de code construisant un terme de preuve

Inconvénient : un gros terme de preuve à chaque utilisation (ex. Omega)

Réflexion : tactique écrite et certifiée dans Coq

L'exécution de la tactique est alors effectuée par la *réduction* de Coq

Le terme de preuve est très réduit (car factorisé dans le résultat de correction de la tactique réflexive)

Réflexion : principe

Première étape : les termes réifiés

```
Inductive prop : Set :=
  | Ptrue : prop
  | Pfalse : prop
  | Pand : prop -> prop -> prop
  | Por : prop -> prop -> prop.
```


Réflexion : principe (suite)

Interprétation des termes réifiés

```
Fixpoint interp [p:prop] : Prop :=
  Cases p of
  | Ptrue => True
  | Pfalse => False
  | (Pand p1 p2) => (interp p1) /\ (interp p2)
  | (Por p1 p2) => (interp p1) \/ (interp p2)
  end.
```


Réflexion : principe (suite)

Deuxième étape : la procédure de décision écrite en Coq

```
Fixpoint decide [p:prop] : bool :=
  Cases p of
  | Ptrue => true
  | Pfalse => false
  | (Pand p1 p2) => (andb (decide p1) (decide p2))
  | (Por p1 p2) => (orb (decide p1) (decide p2))
  end.
```


Réflexion : principe (suite)

Dernière étape : résultat de correction

```
Theorem decide_correct : (p:prop) (decide p)=true -> (interp p).
Proof.
Induction p; Simpl; Auto.
Discriminate.
Intro p0; Case (decide p0); Simpl; Auto Orelse Intros; Discriminate.
Intro p0; Case (decide p0); Simpl; Auto Orelse Intros; Discriminate.
Save.
```

Réflexion : principe (suite et fin)

Application :

```
Goal P.
Change (interp p).
Apply decide_correct.
Reflexivity.
```

Exemple :

```
Goal (True /\ False) \/ (True \/ False).
Change (interp (Por (Pand Ptrue Pfalse) (Por Ptrue Pfalse))).
Apply decide_correct; Reflexivity.
```


Réflexion : gain

Le terme de preuve se réduit à

`(decide_correct p (refl_equal bool true))`

En pratique, le calcul de p (réification) est fait par une tactique ML

La tactique `Quote` l'automatise lorsque la fonction d'interprétation est assez simple

Réflexion : quelques exemples

- `Ring`
- `ROmega`
- `BDDS` (contribution de Kumar Neeraj Verma)

Certification de programmes impératifs

Coq est livré avec une tactique de certification de programmes impératifs, `Correctness`

Interprète les programmes impératifs par des programmes fonctionnels, puis utilise la tactique `Refine` de Coq

`Correctness` *id pgm-annoté*. $\Rightarrow$ obligations de preuve (buts Coq)

L'apparence de la logique de Hoare (sans en être)

Programmes impératifs

Syntaxe à la Caml, avec annotations à la Hoare

- pré- et post-conditions (optionnelles) : `{`P`}` e `{`Q`}`
- références et tableaux : `!`x / x `:=` e / `ref` e / t`[`e`]` / t`[`e_1`] :=` e_2
- séquence : `begin` e_1`;` ...`;` e_n `end`
- conditionnelle : `if` e_1 `then` e_2 `else` e_3
- boucle : `while` e_1 `do { invariant` I `variant` ϕ `}` e_2 `done`
- variable locale : `let` x `=` e_1 `in` e_2
- fonction récursive : `let rec` f `:` τ `{ variant` ϕ `} =` e
- types de données : définis dans Coq + syntaxe usuelle pour `Z`

Utilisation

```
Require Correctness.

Global Variable x : Z ref.

Correctness p
  { `0 <= x < 10` }
  begin x := !x + 1; if !x = 10 then x := 0 end
  { `0 <= x < 10` }.
```

On obtient deux buts Coq:

```
x : Z
Pre1 : `0 <= x < 10`
Test1 : `x+1 = 10`
============================
`0 <= 0 < 10`
```

```
x : Z
Pre1 : `0 <= x < 10`
Test1 : `x+1 <> 10`
============================
`0 <= x+1 < 10`
```


Exemples de programmes certifiés

- tris : insertion, quicksort et heapsort en place
- Knuth-Morris-Pratt
- Bresenham (tracé de droite à l'écran)
- distance d'édition
- algorithme de Floyd
- racine carrée efficace (GMP)

Tactique Certifiée

Bruno Barras & Jean-Christophe Filliâtre

Décider de la relation `moves`

Le but de l'exercice est de prouver la décidabilité de la relation `moves` sur les damiers.

Pour cela on utilise une fonction de normalisation des damiers en retournant les lignes et les colonnes de manière à mettre uniquement des blancs sur la première ligne et colonne.

Invariant

On appelle `invariant` une fonction `f` sur les damiers telle que `(moves b1 b2)`$\rightarrow$`(f b1)=(f b2)`

Montrer que si `f` est un invariant
et `~(f b1)=(f b2)`
alors `~(moves b1 b2)`.

Involution

On appelle `involution` une fonction telle que `(f (f x))=x`.

On pourra supposer sans démonstration que `return_line` et `return_col` sont des involutions.

On en déduit que les relations `move` et `moves` sont symétriques.

Normalisation

Définir une fonction `force_white`:`board` $\rightarrow$ `board` qui retourne ligne et colonnes d'un damier pour mettre des blancs sur la première ligne et la première colonne.

Montrer que `(force_white b1)=(force_white b2)` $\leftrightarrow$`(moves b1 b2)`

Décidabilité de `moves`

En utilisant la décidabilité de l'égalité sur les damiers, énoncer puis démontrer la décidabilité de la relation `moves`.

Construire à partir de cette preuve un programme :
`dec` de type `board` → `board` → `bool`.

Prouver que si ce `(dec b1 b2)=true` alors `(moves b1 b2)`
et si `(dec b1 b2)=false` alors `~(moves b1 b2)`

En déduire des preuves "automatiques" pour les résultats établis dans la première partie.