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)