Valeurs certifiées

Yves Bertot

Introduction

- Utiliser les types inductifs pour garantir des propriétés
- Ajouter du sens à des valeurs booléennes
- Construire des paires dépendantes

Mêler données et preuves

- Un constructeur d'un type inductif peut avoir des arguments qui sont des preuves
- Un objet obtenu par ce constructeur contient un certifcat
- Le certificat doit être fourni à la construction
- Il peut être récupéré par filtrage

Racine carrée

- `Inductive sqrt_data (n : nat) : Type :=`
  `csd : forall m, m * m <= n < (m + 1) * (m + 1) ->`
  `sqrt_data n.`
- Une fonction de calcul de racine carrée aura le type
  `forall n, sqrt_data n`
- Pour construire une telle fonction il faut
  1. savoir trouver une racine carrée de `n`
  2. savoir prouver que c'est bien une racine

Filtrage sur une valeur avec certificat

- Le filtrage sur une valeur avec certificat fait apparaître pour chaque constructeur
  1. Les données qu'il contient
  2. Les preuves de propriétés garanties sur ces données
- Supposons que `t` a le type `sqrt_data n`
- Dans l'expression de filtrage

  `match t with csd m h => ...end`

  `m` a le type `nat` et `h` est une preuve de

  `m * m <= n < (n + 1) * (n + 1)`

Programmation par preuve

- Pour produire des valeurs certifiées, les filtrages doivent être dépendants
- Plus faciles à écrire en tant que preuves
- Suivre la structure des algorithmes à l'aide de tactiques
  - construction de fonction: `intros`
  - traitement par cas: `case`
  - appel de fonction: `apply`
  - récursion: `elim` ou `induction`

Valeurs booléennes annotées

- `Inductive sumbool (A B : Prop) : Type :=`
  `left : A -> sumbool A B`
  `| right : B -> sumbool A B.`
- Notation `{A}+{B}` pour `sumbool A B`
- Les valeurs de `{A}+{B}` se construisent comme les preuves de `A \/ B`
  - Pas droit au traitement par cas sur `C \/ D`
- Au traitement par cas, deux cas sont créés, le premier avec `A` comme hypothèse, le second avec `B`
  - Ajoute de l'information dans le contexte
- Dans les librairies Coq, voir les fonctions `_dec`

Paires dépendantes

- Approche générique pour associer une valeur et une preuve
- `Inductive sig (A : Type) (P : A -> Prop) : Type :=`
  `exist : forall x: A, P x -> sig A P.`
- L'argument `A` est implicite pour `sig` et `exist`
- Notation `{x : A | P x }` pour `sig (fun x : A => P x)`
- Se manipule comme une quantification existentielle
- Pour construire une valeur, fournir un témoin et prouver la propriété
- Au traitement par cas, récupérer la valeur et la preuve.

Programmation par preuve

Yves Bertot

Introduction

- Utiliser des tactiques pour programmer
- Fabriquer des fonctions qui contiennent des preuves
- Gérer les types dépendants

Exemple: racine carrée

- `Require Import Arith.`
- `Inductive sqrt_data (n : nat) : Type :=`
  `csd : forall m, m * m <= n < (m + 1) * (m + 1) ->`
  `sqrt_data n.`
- Première étape: construire une fonction de type
  `forall n m, n < m * m -> sqrt_data n`
- Utiliser ce type de fonction comme un énoncé logique
- `Definition srec :`
  `forall n m, n < m * m -> sqrt_data n.`
- Utilisation du mot clef `Definition` au lieu de `Lemma`.

Recherche de la racine carrée

- Regarder si le prédécesseur est encore plus grand
  1. si oui, recommencer avec le prédécesseur
  2. si non, le prédécesseur est la racine carrée
- Ceci indique un procédé par récurrence: tactique `induction`

```
intros n m; induction m.
...
  n : nat
  ============================
   n < 0 * 0 -> sqrt_data n

subgoal 2 is:
n < S m * S m -> sqrt_data n
```

Traitement du cas de base

```
============================
 n < 0 * 0 -> sqrt_data n
```

- La prémisse est incohérente
- Recherche d'un théorème

  ```
  SearchPattern (~ _ < 0).
  lt_n_0: forall n :  nat, ~ n < 0
  ```

- `intros abs; elim (lt_n_0 _ abs).`

Traitement du cas récursif

```
...
  IHm :  n < m * m -> sqrt_data n
  ============================
   n < S m * S m -> sqrt_data n
```

- L'hypothèse `IHm` correspond à un appel récursif
- Elle ne peut s'appliquer que si `n < m * m`
- Il faut faire le test de comparaison!

  ```
  SearchPattern ({_ <= _}+{_}).
  le_lt_dec : forall n m : nat, {n <= m}+{m < n}
  ```

- traiter la valeur

  `le_lt_dec (m * m) n : {m * m <= n}+{n < m * m}`

  comme une disjonction

Introduction d'un test

- Test d'une valeur de type `sumbool`
- `destruct (le_lt_dec (m * m) n) as [sqle | nltsq].`
- Deux nouveaux buts avec les hypothèses venant du test

```
  sqle : m * m <= n
  ============================
   n < S m * S m -> sqrt_data n

subgoal 2 is:
n < S m * S m -> sqrt_data n
```

- Dans le premier cas on a assez pour conclure

Construire la valeur certifiée

- Appeler le constructeur du type
- `intros h; apply (csd n m).`

```
...
  sqle : m * m <= n
  h : n < S m * S m
  ============================
   m * m <= n < (m + 1) * (m + 1)
replace ((m + 1) * (m + 1)) with (S m * S m) by
ring.
...
  ============================
   m * m <= n < S m * S m
split; assumption.
```

Dernier cas

- Si `n` est plus petit `m * m`, on peut faire un appel récursif
- ...

```
  IHm : n < m * m -> sqrt_data n
  nltsq :  n < m * m
  ============================
   n < S m * S m -> sqrt_data n
intros _; apply IHm; exact nltsq.
Proof completed.
Defined.
```

- Utiliser `Defined` pour sauver la valeur.

Exemple de programmation avec valeurs certifiées

Yves Bertot

Introduction

- Définir une fonction de calcul de racine carrée
- Utiliser un type de valeurs certifiées.
- Utiliser des fonctions prédéfinies à valeurs certifiées

Définir le type de valeur certifiée

- `Inductive sqrt_data (n : nat) : Type :=`
  `csd : forall m, m * m <= n < (m + 1) * (m + 1) ->`
  `sqrt_data n.`
- Ce type décrit les racines carrées de `n`
- Il dépend de `n`
- Le constructeur a deux arguments, dont le deuxième est une preuve

Recherche de la racine carrée

- Commencer avec une valeur plus grande que la racine carrée
- Regarder si le prédécesseur est encore plus grand
  1. si oui, recommencer avec le prédécesseur
  2. si non, le prédécesseur est la racine carrée

```
Fixpoint srec n x : n < x * x -> sqrt_data n :=
match x return n < x * x -> sqrt_data n with
  0 => fun h : n < 0 * 0 =>
         False_rec (sqrt_data n) prf1
| S p => fun h => match tester si p * p <= n with
           true => csd p prf2
         | false => srec n p prf3
         end
end.
```

Détermination des preuves

- Les preuves sont des justifications raisonnables sur l'algorithme
- `prf1` doit être une preuve de `False`
  - Utiliser le fait que `n < 0 * 0` n'est pas possible
  - On aurait du trouver la racine avant!
- `prf2` doit être une preuve de
  `p * p <= n < (p + 1) * (p + 1)`
  - La première partie doit venir du test
  - La seconde partie doit venir de la condition initiale
    dans le contexte: `h : n < S p * S p`
- `prf3` doit être une preuve de `n < p * p`
  - A cet endroit il y a un appel récursif sur `p`
  - L'information doit venir du test

Test avec information logique

- Dans les deux branches du test de comparaison, on a besoin d'informations
- Utilisation d'une fonction prédéfinie avec type `sumbool`
- `Require Import Arith.`
- `SearchPattern ({_ <= _}+{_}).`
  `le_lt_dec : forall n m : nat, {n <= m}+{m < n}`

```
Fixpoint srec n x : n < x * x -> sqrt_data n :=
match x return n < x * x -> sqrt_data n with
  0 => fun h : n < 0 * 0 =>
         False_rec (sqrt_data n) prf1
| S p => fun h => match le_lt_dec (p*p) n with
           left h1 => csd p prf2
         | right h2 => srec n p prf3
         end
end.
```

Construction des preuves

- Pour `prf1`, utiliser `lt_n_O : forall n, ~n < 0`
  - Se souvenir que `~A` ≡ `A -> False`
- Pour `prf2`, prouver un lemme auxiliaire

```
Lemma th2 : forall n p,
n < S p * S p -> p * p <= n ->
    p * p <= n < (p + 1) * (p + 1).
intros n p.
replace ((p+1)*(p+1)) with (S p * S p) by ring.
intros; split; assumption.
Qed.
```

- Pour `prf3`, le test donne directement `h2 : n < p * p`

Fonction srec complète

```
Fixpoint srec n x : n < x * x -> sqrt_data n :=
  match x return n < x * x -> sqrt_data n with
    0 => fun h => False_rec (sqrt_data n) (lt_n_0 n h)
  | S p => fun h => match le_lt_dec (p*p) n with
      left h1 => csd n p (th2 n p h h1)
    | right h2 => srec n p h2
    end
  end.
```

Fonction principale

- Appeler la fonction récursive avec une bonne valeur initiale
- Choisissons `S n` et prouvons la bonne propriété

```
Lemma prf5 : forall n, n < S n * S n.
intros; simpl; generalize (n * S n); intros x;
omega.
Qed.

Definition sqrt n : sqrt_data n :=
   srec n (S n) (prf5 n).
```

- Aussi possible de construire le programme comme une preuve

Extraction

Yves Bertot

Introduction

- Utiliser Coq pour produire des programmes
- Chaîne de production passant par `OCaml`
- Oubli des information “non algorithmiques”
- Ce cours: une approche possible, mais pas entièrement fidèle

Enlever les informations logiques

- Tout type de Coq peut être associé à un type `OCaml`
- Si le type est une proposition ou une sorte, `unit`
- Pour les types inductifs
  - garder le nombre de constructeurs
  - enlever les champs preuves

Extraction du type sumbool

- `Inductive sumbool (A B : Prop) : Type :=`
  `left : A -> sumbool A B`
  `| right : B -> sumbool A B.`
- Oublier les preuves, c'est diminuer le nombre d'arguments des constructeurs
- `type sumbool = Left | Right;;`
- Ce type énuméré à deux constructeurs est équivalent à `bool`
- Associer à `bool` est possible sur directive de l'utilisateur
  - `Extract Inductive sumbool =>`
    `"bool" [ "true" "false" ].`

Extraction des paires dépendantes

- Dans une paire dépendante, un seul constructeur avec un seul argument informatif
- Le type est directement associé au type du champ informatif
- `{x : nat | P x}` est associé à `nat`

Extraction des appels de fonctions

- Extraction des fonction Coq vers des fonctions Caml de même nom
- Les arguments de preuve sont remplacés par `()`
- Le nombre d'arguments des fonctions est préservé
- Traitement différent quand les fonctions sont des constructeurs
  - Syntaxe différente en Ocaml
  - Disparition des arguments preuves
  - Parfois disparition du constructeur (paire dépendante)
- `pred_safe :`
  `forall x, x <> 0 -> {y : nat | x = S y}`
  - en Ocaml : `pred_safe : nat -> unit -> nat`
  - `pred_safe 3 (sym_not_eq (O_S 2))` est extrait en `pred_safe 3 ()`

Extraction du filtrage

- Tenir compte de la disparition des champs de preuve
- Preuves non manquantes car remplacées par `()` dans le corps
- Filtrages dépendants remplacés par des filtrages non dépendants
- Cas particuliers
  - filtrage sur l'égalité: changement de type dépendant, ignoré
  - filtrage sur `False`: remplacé par une exception

Exemple de fonction à type dépendant

```
Definition pred_safe x := x <> 0 -> {y | x = S y} :=
match x return x <> 0 -> {y | x = S y} with
  0 => fun h : 0 <> 0 =>
       False_rec {y | 0 = S y} (h (refl_equal 0))
| S p => fun _ =>
       exists (fun y => S p = S y) p (refl_equal (S p))
end
```

- Extraction

```
let pred_safe = function
| 0 -> assert false
| S p -> p
```

Conclusion

- Ceci n'est qu'une approximation
- En réalité, le nombre d'arguments des fonctions change
- Utilisable pour de gros développements, e.g. compilateur