Familles de types récursifs

Yves Bertot

Introduction

- Utiliser les paramètres de types inductifs
- Ajouter une faible dose de dépendance
- décrire le polymorphisme

Définir un type dans une section

- Dans une section Coq on peut avoir des variables locales
- Définitions de types après les variables locales
- Comportement après fermeture
  - la variable locale devient un paramètre de la définition
- Modélisation directe par les paramètres dans les définitions

Familles polymorphes de types

- Utilisation simple de la dépendance
- ```
  Inductive list (A : Type) : Type :=
      nil : list A
    | cons : A -> list A -> list A.
  ```
- `list` n'est pas un type
  - `list nat`, `list bool`, `list Z` sont des types
  - `list (list nat)` est un type
  - `list : Type -> Type`, c'est une *famille de types*
- Le constructeur `cons` a un type dépendant

  `cons : forall A : Type, A -> list A -> list A`
- La récursion ne fait appel qu'à des objets de la même famille
  ```

Arguments implicites dans le type des listes

- En théorie, `cons` est une fonction à trois arguments
- Le premier argument est un type
- En pratique, il existe un mécanisme d'*arguments implicites*
- Le premier argument n'est pas donné
  - `cons 1 (cons 3 nil)) :  list nat`
- Même chose pour `nil : forall A : Type, list A`
- Il est possible de forcer les arguments explicites en ajoutant le préfixe `@` devant la fonction
  - `@cons nat : nat -> list nat -> list nat`

Autres types polymorphes

- Le type *produit cartésien* des couples
- `Inductive prod (A B : Type) : Type :=`
  `pair : A -> B -> prod A B.`
- Coq fournit deux notations:
  - `A * B` pour `prod A B`
  - `(x, y)` pour `pair x y`
  - Les deux premiers arguments de `pair` sont implicites
- Le type *valeur optionnelle*
- `Inductive option (A : Type) : Type :=`
  `Some : A -> option A | None : option A.`

Familles de types récursifs hétérogènes

Yves Bertot

Introduction

- mêler types récursifs et dépendants
- Assurer la cohérence de données
- associer des informations logiques

Familles dépendantes hétérogènes

- ```
  Inductive fin : nat -> Type :=
      new : forall n, fin (S n)
    | inj : forall n, fin n -> fin (S n).
  ```
- le type `fin 0` est vide
- le type `fin (S n)` contient:
  - les images de `fin n` par `inj n`
  - un élément supplémentaire apporté par `new n`
- Le type `fin n` contient exactement `n` éléments
  ```

Autres types dépendants hétérogènes

- Un type de listes de longueur fixée, aussi appelées *vecteurs*
- ```
  Inductive vector (A : Type) : nat -> Type :=
      vnil : vector A 0
    | vcons :
       forall n, A -> vector A n -> vector A (S n).
  ```
- Un type d'arbres parfaitement équilibrés (utilisés dans la transformée de Fourier rapide)
  ```

Familles avec des éléments vides

- Les familles avec éléments vides ont un intérêt logique
- ```
  Inductive eq (A : Type) (x : A) : A -> Prop :=
      refl_equal : eq A x x.
  ```
- L'argument `A` est implicite
- Notation `x = y` pour `eq x y`
- Le type `x = y` est habité exactement lorsque `y` est `x`
- La tactique `reflexivity` fait `apply refl_equal.`
  ```

Solution alternative pour les vecteurs

- ```
  Inductive vecteur' (A : Type) (n : nat) : Type :=
  Vecteur' :
    forall l : list A, length l = n -> vecteur' A n.
  ```
- Passage d'un vecteur à une liste et vice-versa aisé
- Oubli de l'information de longueur
- Programmation récursive plus facile
  ```

Un type avec récursion et information logique

- ```
  Inductive even : nat -> Type :=
    ev0 : even 0
  | ev2 : forall n, even n -> even (S (S n)).
  ```
- Les types `even 0`, `even 2`, `even 4` ont des éléments
- Les types `even 1`, `even 3`, n'en ont pas
- Les éléments ne sont pas très importants
- L'information d'existence est importante du point de vue logique
  ```

Principes de récurrence pour les prédicats inductifs

Yves Bertot

Introduction

- Comprendre la structure des principes de récurrence
- Etudier le cas particulier des propositions inductives
- Observer l'usage pratique

Détermination du prédicat principal

- Exprimer qu'une propriété est satisfaite pour tous les éléments de tous les éléments de la famille
- Définir un prédicat qui couvre
  1. tous les arguments possibles du prédicat inductifs,
  2. tous les éléments possibles pour un choix d'arguments,
- `Inductive fin : nat -> Type := ...`
- Quantifier sur un prédicat

  `P : forall n : nat, fin n -> Prop`

Traitement des constructeurs

- Traiter chaque constructeur
- La propriété doit être satisfaite pour tous les arguments possibles
- `new : forall n : nat, fin (S n)`
- Donne `forall n : nat, P (S n) (new n)`
- Les références récursives amènent des hypothèses de récurrence
- `inj : forall n : nat, fin n -> fin (S n)`
- Donne

```
forall n : nat, forall t : fin n, P n t ->
     P (S n) (inj n t)
```

Epilogue du principe de récurrence

- Exprimer que la propriété est satisfaite dans tous les cas
- `Inductive fin  : nat -> Type := ...`
- Donne `forall (n : nat) (t : fin n), P n t`

Paramètres dans les principes de récurrence

- Les paramètres ne changent pas dans l'utilisation du type
- La quantification sur les paramètres arrive en premier
- ```
  Inductive list (A : Type) : Type :=
      nil : list A | cons : A -> list A -> list A.
  ```
- Donne

```
forall (A : Type) (P : list A -> Prop),
    P nil ->
    (forall a l, P l -> P (cons a l)) ->
    forall (l : list A), P l
```
```

Principes de récurrence pour les prédicats

- Considérer que les preuves n'ont pas d'importance
- Enlever l'argument du type dans le prédicat à prouver
- Si le type inductif `T` a le type `A -> Prop`, alors le prédicat a le type `A -> Prop`
  - au lieu de `forall (a : A), T a -> Prop`
- ```
  Inductive eq (A : Type) (x : A) : A -> Prop :=
      refl_equal : eq A x x.
  ```
- Donne

```
forall (A : Type) (x : A) (P : A -> Prop),
    P x ->
    forall (y : A), x = y -> P y
```
```

Preuve par récurrence et réécriture

- Le principe de récurrence de l'égalité exprime le remplacement de `y` par `x` dans la propriété à prouver
- Plus généralement, remplacement de variables par les valeurs apparaissant dans les constructeurs

Présentation inductive de concepts usuels

Yves Bertot

Introduction

- Les types inductifs sont un “couteau suisse” de la logique
- Connecteurs logiques
- égalité
- Ordre sur les nombres naturels

Conjonction

- `Inductive and (A B : Prop) : Prop :=`
  `conj : A -> B -> and A B.`
- Notation `A /\ B` pour `and A B`
- Le constructeur indique que pour prouver `A /\ B` il suffit de fournir des preuves de `A` et `B`
- Principe de récurrence
  `forall A B P : Prop, (A -> B -> P) -> A /\ B -> P`
- Nouveau but : prouver `P` sachant `A` et `B` séparément
- Ceci explique pourquoi `elim`, `case`, `destruct` s'utilisent sur des hypothèses conjonctives

L'égalité

- `Inductive eq (A : Type) (x : A) : A -> Prop :=`
  `refl_equal : eq A x x.`
- Argument `A` implicite, notation `x = y` pour `eq x y`
- Principe de récurrence

```
forall (A : Type) (x : A) (P : A -> Prop),
    P x ->
    forall (y : A), x = y -> P y
```

- Exprime le remplacement de `y` par `x` dans l'énoncé à prouver

Ordre sur les entiers

- ```
  Inductive le (n : nat) : nat -> Prop :=
      le_n : le n n
    | le_S : forall m, le n m -> le n (S m).
  ```
- `n` est paramètre
- Notation `n <= m` pour `le n m`
- Principe de récurrence

  ```
  forall (n : nat) (P : nat -> Prop)
      P n ->
      (forall m, P m -> P (S m)) ->
      forall m, n <= m -> P m
  ```
- Preuve par récurrence sur `n <= m`
  - 1 cas: remplacement de `m` par `n`
  - 1 cas: remplacement de `m` par `S m`,
    mais hypothèse que `n <= m` et hypothèse de récurrence
  ```

Autres utilisations

- Programmes Prolog
- Relations récursives
- Sémantique d'un langage de programmation
- Enregistrements et structures mathématiques