Les types dépendants

Yves Bertot

Familles de types

- Représenter des collections de types A_1, A_2, ...
  - Exemple : tableaux de longueur 1, 2, ...
  - Aussi : nombres inférieurs à 1, 2, ...
- Utiliser des fonctions de type `nat -> Type` pour représenter ces familles
  - `Zarray : nat -> Type`
  - `bound : nat -> Type`
- Choix arbitraire pour le type des indices
  - Pas nécessairement un type de nombres
  - N'importe quel type déjà défini

Fonctions à type dépendant

- Possibilité de définir des fonctions sur les types indicés
  - `read2 : Zarray 2 -> bound 2 -> Z`
  - permet par exemple de décrire l'accès sans erreur entre les bornes d'un tableau
- Suggère l'existence de familles de fonctions à type indicé
  - `read3 : Zarray 3 -> bound 3 -> Z`
  - `read4 : Zarray 4 -> bound 4 -> Z`
  - `...`
- La famille de fonctions peut directement être représentée comme une fonction
  - Si `i` est un indice de type nat `read i` doit avoir le type `Zarray i -> bound i -> Z`
- Besoin d'une notation nouvelle pour le type de cette nouvelle fonction
  - `read : forall i:nat, Zarray i -> bound i -> Z`

Le type *produit dépendant*

- Notation `forall x : A, E x`
- Décrit le type d'une fonction qui attend un argument de type `A`
- Le type de retour dépend de la valeur en entrée
- si `f` a le type `forall x : A, E x`, si l'expression `b` a le type `A`, alors l'expression `f b` a le type `E b`
- Exemple avec `read`
  - `read : forall i:nat, Zarray i -> bound i -> Z`
  - `read 2 : Zarray 2 -> bound 2 -> Z`

Bonne formation du produit dépendant

- Conditions pour que l'expression `forall x : A, E x` soit bien formée
- Il faut que `A` soit un type (de type `Set`, `Type`, `Prop`),
- Il faut que `E x` soit un type (de type `Set`, `Type`, `Prop`) dès que `x` de type `A`
  - Si `Zarray` et `bound` ont le type `Type`
  - Dès que `i` a le type `nat`, `Zarray i` et `bound i` ont le type `Type`,
  - Alors `Zarray i -> bound i -> Z` a le type `Type`,
  - Alors `forall i : nat, Zarray i -> bound i -> Z` est bien formé
  - `forall i : nat, Zarray i -> bound i -> Z` a le type `Type`

Le polymorphisme

Yves Bertot

Fonctions polymorphes

- Le type des fonctions polymorphe s'exprime avec un produit dépendant
- Exemple de la fonction identité:
  `Definition id := fun (A : Type) (x :  A) => x.`
- `Check id.`
- Le système répond `id :  forall A : Type, A -> A`

Types de données polymorphes

- Le type des listes est polymorphe en Coq:
- `list :  Type -> Type`
- les constructions de base pour les listes sont polymorphes:
- `nil :  forall A:Type, list A`
- `cons :  forall A: Type, A -> list A -> A`
- Attention, le premier argument de `nil` et `cons` est implicit.

Fonctions à types dépendants

Yves Bertot

Introduction

- Des fonctions à valeur dans des types différents
- Des familles de types
- Une nouvelle façon de noter le type des fonctions

Familles de types

- Type d'indices `I`
- famille de type A_i représentée par une fonction `A` de type `I -> Type`
- A_i est représenté par `A i`
- Nouveauté: considérer une fonction `f` dont le type de départ est `I` et telle que `f i` a le type `A i` pour tout `i` dans `I`
  - Le type d'arrivée n'est pas toujours le même
  - Le type d'arrivée dépend de la valeur en entrée

Nouvelle notation de type de fonction

- Notation : `f : forall i : I, A i`
- si `x` est une valeur de type `I`,
  `f x` a le type `A x`
- Même notation que pour les formules logiques
- Aussi appelé *produit dépendant*
- Généralisation du produit cartésien

Généralisation du produit cartésien

- un multiplet $\texttt{(x,y,z)} : \texttt{A}_1 * \texttt{A}_2 * \texttt{A}_3$
- Considérons la fonction de projection $\pi_n(\texttt{x}, \texttt{y}, \texttt{z})$
  - si $n = 1$ elle retourne $\texttt{x} : \texttt{A}_1$
  - si $n = 2$ elle retourne $\texttt{x} : \texttt{A}_2$
  - si $n = 3$ elle retourne $\texttt{x} : \texttt{A}_3$
- Le multiplet est comme une fonction, mais son type est dépendant

Exemple de famille de type

- Si `P : A -> Prop` est un prédicat
- Coq permet de définir le type `{x : A | P x}`
- Par exemple :

```
Definition divides (x y : nat) :=
    exists k, x = k * y.
```

- on peut définir la famille de types:
  `fun y => {x : nat | divides x y}`
- `{x : nat | divides x 134}` contient tous les multiples de 134
- C'est un type: définir ce type ne veut pas dire que l'on sait trouver tous les multiples de 134
- le type `{x : nat | divides x 134}` est différent de `nat`

Exemples d'utilisation : protection contre les débordements

- `Definition bound n := {x | x < n}.`
  le type des entiers entre 0 et n
- `array_Z n` le type des tableaux de taille n
- Type d'une fonction d'accès :
  `forall n, bound n -> array_int n -> Z`
- Evite les erreurs de dépassement de tableau
- Détection des erreurs au moment de la vérification des types
  - pas d'erreur à l'exécution

Exemple d'utilisation: Spécifications de programmes

- Le type `{x | P x} -> {y | Q y}` est riche en informations
  - Cette fonction n'accepte que des arguments qui satisfont `P`
  - Cette fonction ne produit que des résultats qui satisfont `Q`
- le type `forall x, {y | R x y}` peut spécifier le comportement
  - fournir une fonction de ce type, c'est garantir que l'on sait produire pour chaque `x` un `y` qui satisfait `R x y`
  - Si `R` est tel que `y` est unique la valeur retournée est entièrement spécifiée

Les propositions sont des types

Yves Bertot

Introduction

- Propositions vues comme des type de preuves
- Utilisation dans des fonctions dépendantes
- Utilisation avec la notation de sous-type

Types de preuves

- Si `P` est une proposition, c'est une collection dont les éléments sont des preuves de `P`
- Si `P` est une proposition fausse, il n'y a pas d'élément
- `P -> Q` est le type des fonctions qui expliquent le lien entre les preuves de `P` et les preuves de `Q`
- Si `P : Prop`, on peut écrire `fun x : P => ...`, `P -> nat`, `forall x : P, ...`
- Si `P : I -> Prop`, le type `forall i, P i` dit que `P` est toujours satisfait
- une preuve de `forall i, P i` est un procédé qui indique comment obtenir `P i` pour n'importe quel `i`
  - Si `th : forall i : I, P i` et `a : I`, alors `th a : P a`

Utilisation dans les spécifications

- Décrire les fonctions qui ne peuvent être utilisées que pour des arguments qui satisfont `P`
  - `forall x : A, P x -> ...`
- Exemple pour la fonction de division: éviter la division par 0
  - `nat -> forall y, y <> 0 -> nat * nat`
  - type minimal pour éviter la division par 0
- Spécification plus précise
  - ```
    forall x y, y <> 0 ->
        {p : nat * nat |
            let (q,r) := p in x = q * y + r /\ r < y}
    ```
    ```

Construire des preuves comme des fonctions

- Première méthode: Combiner les théorèmes comme des fonctions
- Exemple : `forall n, n <= S n`
  - Utiliser `le_S : forall n m, n <= m -> n <= S m`
    `le_n : forall n, n <= n`
- `fun n => le_S n n (le_n n) : forall n, n <= S n`

Utilisation de exist

- `{x : T | P x}` est une notation pour
  `sig (fun x : T => P x)`
- Fonction existante de Coq
  `exist : forall (A : Type)(P : A -> Type)(x : A), sig P`
  - L'argument `A` est implicite
- Théorème existant:
  `refl_equal : forall (A : Type)(x : A), x = x`
  - A est un argument implicit, `refl_equal 0 : 0 = 0`
- `Definition v0 :`
  `{b : bool | if b then 0=0 else 0<>0} :=`
  `exist _ true (refl_equal 0).`

Utilisation de nat_rect

- ```
  nat_rect : forall P : nat -> Type,
      P 0 -> (forall n, P n -> P (S n)) ->
      forall n, P n
  ```
- ```
  nat_rect
      (fun n =>
          {b : bool | if b then n=0 else 0<>n})
      v0
  ```
  est bien formé

Construire les fonctions dépendantes comme des preuves

- Poser le type de la fonction comme un énoncé de théorème
- utiliser les tactiques pour construire l'intérieur de la fonction
- `elim` est équivalent à l'utilisation de `..._rect`
- `case` correspond à une construction de filtrage
- `apply` correspond à l'appel d'une fonction
- `exact` permet de donner la valeur retournée
- `exists` correspond à l'appel de la fonction `exist`

Filtrage et récursion dépendants

Yves Bertot

Introduction

- Construction de filtrage avec dépendance
- Récursion avec dépendance
- Construction de principes de récurrence

Filtrage avec dépendance

- Par défaut, pas de dépendance dans les constructions de filtrage `match` e `with` p_1 `=>` v_1 `|` p_2 `=>` `...end`
- Dépendance explicite ajoutée avec une clause `as ...return ...`
- Chaque règle retourne dans un type qui dépend du motif
- Compliqué à écrire, mais utilisation possible de tactiques

Exemple de filtrage dépendant

- suppose l'existence de `O_S : forall n, O <> S n`
- `or_introl : forall A B, A -> A \/ B`
- `or_intror : forall A B, B -> A \/ B`
-
```
Definition ex1 : forall n, O = n \/ O <> n :=
fun n =>
match n as x return O = x \/ O <> x with
| O => @or_introl (O = O) (O <> O) (refl_equal O)
| S p => @or_intror (O = S p) (O <> S p) (O_S p)
end.
```

Filtrage dépendant générique

- Même approche, mais avec une famille de types générale
- ```
  Definition ex_gen (P : nat -> Type)
      (V0 : P 0) (fs : forall n, P (S n)) (n:
  nat) : P n :=
  match n return P n with
  | 0 => V0
  | S p => fs p
  end.
  ```
  ```

Ajout de la récursion

- Si un motif de filtrage a la forme `C ...v ...`
  où `C` est un constructeur
- Les appels récursifs sont autorisés sur `v`
- Le type de l'appel récursif dépend de `v`

Définition de nat_rect

- ```
  Fixpoint nat_rect (P : nat -> Type)
  (V0 : P 0) (fs : forall n, P n -> P (S n))
  (n : nat) : P n :=
  match n return P n with
  | 0 => V0
  | S p => fs p (nat_rect p)
  end.
  ```
- Fonction similaire `nat_ind` où `Prop` remplace `Type`
- ```
  nat_ind : forall P : nat -> Prop,
      P 0 -> (forall n, P n -> P (S n)) ->
  forall n, P n
  ```
- Construit automatiquement par Coq, utilisé par la tactique `elim`

récurrence spécialisée

- Principe de récurrence adapté pour un schéma de récursion
- exemple pour la récursion en 2 étapes sur `nat`
-
```
Fixpoint nat_ind2 (P : nat -> Prop)
(V0 : P 0) (V1 : P 1)
(fs : forall n, P n -> P (S (S n)))
(n : nat) : P n :=
match n return P n with
| 0 => V0
| 1 => V1
| S (S p) => fs p (nat_ind2 p)
end.
```

tactiques pour le filtrage dépendant

- `case` fabrique un filtrage dépendant par défaut
- `fix` fabrique un contexte d'appel récursif
  - ajoute une nouvelle hypothèse utilisable comme fonction récursive
  - Attention aux restrictions d'usage
- `refine` permet de mélanger les genres

Types récursifs dépendants

Yves Bertot

Introduction

- Utilisation de types inductifs où les constructeurs ont des types dépendants
- Définitions inductives de familles de types
- Adaptation du filatrage et des principes de récurrence

Constructeurs à type dépendant

- Imposer des propriétés de cohérence aux données
- `Inductive t4 : Type :=`
  `ct4 : forall n : nat, n<>0 -> t4.`
- Pour construire un élément du type, il faut trouver des composantes cohérentes
- Filtrer pour récupérer les composantes et leur cohérence

Filtrage sur un type à constructeurs dépendants

- Permet de récupérer en plusieurs éléments les composantes du constructeur
- ```
  fun x:t4 =>
  match x with
  ct4 n p => ...
  end.
  ```
- Dans `... p` est une preuve que `n` est non-nul
- On ne peut pas extraire la deuxième composante sans la première.
  ```

Familles de types inductifs

- Même approche que pour un type inductif, mais définit une fonction
- ```
  Inductive sqrt (n : nat) : Type :=
  csqrt (x : nat)
        (p1 : x * x <= n < (x + 1) * (x + 1)).
  ```
- Tout élément de `sqrt n` contient un `x` qui est l'approximation entière de la racine carrée de `n`
- ```
  Inductive cmp (n m : nat) : Type :=
  cmp1 (_ : n <= m) | cmp2 (_ : m < n).
  ```
- tout élément de `cmp n m` est soit `cmp1 h` lorsque `n <= m`, soit `cmp2 h` lorsque `m < n`
- Le filtrage permet de récupérer les composantes

Cohérence interne des données

- ```
  Inductive t5 (A :Type) : nat -> Type :=
  | t50 : t5 A 0
  | t51 : forall n, A -> t5 A n -> t5 A n -> t5 A (S
  n).
  ```
- Le type `t5 n` décrit des arbres binaires
- La longueur de toutes les branches est égale `n`
- Même hauteur pour les deux sous-arbres d'un noeud composite
  ```

Présentation alternative des arbres binaires équilibrés

- Privilégier les types simples
- Ajouter la cohérence sous forme de fonctions booléennes
- ```
  Inductive bin (A : Type) : Type :=
  L | N (v : A) (t1 t2 : bin A).
  ```
- ```
  Fixpoint
      bal (A : Type)(n:nat)(t : bin A) : bool :=
  match t, n with
  | L, O => true
  | N _ t1 t2, S p => bal A p t1 && bal A p t2
  | _, _ => false
  end.
  ```
- ```
  Inductive t6 (A : Type)(n:nat) : Type :=
  ct6 (t : bin A) (_ : bal A n t = true).
  ```
  ```

Filtrage et récursion sur les familles inductives

- Même fonctionnement que pour les types inductifs simples
- Les fonctions récursives ont plusieurs arguments
  - Les derniers arguments dépendent des premiers
- Traitements différents: paramètres ou non
- Arguments paramètres non-fournis dans le filtrage
  - Ils peuvent être déterminés à partir du contexte

Utilisation de tactiques

- Tactiques `case`, `elim`, `destruct`, `induction`
- Ajoutent de nouvelles hypothèses dont le type dépend du contexte
  - Dépendance sur des valeurs existantes ou sur de nouvelles hypothèses
- Expression de cas pas seulement sur l'argument
  - Les indices sont également touchés
  - Parfois besoin de rendre le but plus général tactiques `revert` et `generalize`

Propositions inductives

Yves Bertot

Introduction

- Lecture logique du type des constructeurs
- Interpretation logique du type inductif
- Utilisation de familles inductifs avec des éléments vides
- Vision concrète des raisonnements
- Traitement par cas et `inversion`

Lecture logique du type des constructeurs

- `Inductive le (n : nat) : nat -> Prop :=`
  `| le_n : le n n`
  `| le_S : forall m, le n m -> le n (S m).`
- Tous les éléments d'un type `le a b` sont obtenus par les constructeurs
- Le type `le 3 4` contient un élément `le_S 3 (le_n 3)`
- Le type `le 1 0` ne contient aucun élément
  - Ni `le_n` ni `le_S` ne pourrait en construire
- En Coq `n <= m` est une notation pour `le n m`

Conception de propriétés inductives

- Les constructeurs doivent être des formules vraies
- La combinaison des constructeurs doit suffir à prouver toutes les instances
- Le principe de récurrence exprime une caractéristique de minimalité
  - Principe de récurrence modifié par rapport aux autres types
  - non-pertinence des preuves
  - Exprime les conditions pour qu'il existe une preuve
    pas les propriétés du terme de preuve

Propriétés inductives usuelles

- Les connecteurs logiques
- La quantification existentielle
- L'égalité
- La contradiction
- Ceci explique pourquoi les tactiques des connecteurs sont `destruct` et `elim`

Connecteur logique

- `Inductive and (A B : Prop) : Prop :=`
  `conj : A -> B -> and A B.`
- Notation `A /\ B` pour `and A B`
- Principe de récurrence associé
  `and_ind : forall A B P : Prop, (A -> B -> P) ->`
  `A /\ B -> P`
- Traitement similaire pour `or` avec 2 constructeurs

Quantification existentielle

- `Inductive ex (A : Type) (P : A -> Prop) : Prop := ex_intro : forall x : A, P x -> ex A P.`
- Notation `exists x:A, P` pour `ex A (fun x => P)`
- Principe de récurrence associé `forall A P Q, (forall x, P x -> Q) -> ex A P -> Q`

Egalité

- `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

Représentation inductive de la contradiction

- `Inductive False : := .`
- Pas de constructeur: jamais prouvable
- Principe de récurrence associé
  `forall P, False -> P`
- Si on a une contradiction on peut construire une valeur dans n'importe quel type