Les fonctions en Coq

Yves Bertot

Les fonctions: outil principal en Coq

- Les fonctions sont les unités de base de la programmation en Coq
- Les fonctions décrivent les opérations à effectuer dans un langage mathématique
- Les fonctions permettent de s'adapter à des données différentes en utilisant des variables
- Les fonctions peuvent elles-mêmes être utilisées comme des données

Construire des fonctions simples

- On définit une fonction en fournissant une expression bien formée et une variable spéciale
  - Utilisez `fun` et => pour indiquer la variable spéciale
- Exemple: la fonction qui ajoute 2 à son entrée
  - `Check fun x => x + 2.`
  - Le système répond
    `fun x : nat => x + 2 : nat -> nat`
  - Le texte "`: nat -> nat`" indique que c'est une fonction qui prend en entrée un nombre et retourne un nombre
  - Coq modifie l'expression pour ajouter un type à la variable `x`
- Les fonctions sont anonymes
  - Comme l'expression `2 + 3`, l'expression `fun x => x + 2` n'a pas de nom
  - Pour donner un nom à une fonction il faut utiliser la commande `Definition`

Appliquer les fonctions

- On utilise une fonction en l'appliquant à une valeur
  - Il suffit d'écrire la fonction sur la gauche de la valeur
  - on n'ajoute des parenthèses que pour éviter les ambiguités, autour de la fonction, ou autour de l'argument
- Exemple: appliquer la fonction qui ajoute 2 au nombre 4,
  - `Eval vm_compute in (fun x => x + 2) 4.`
  - Le système répond `6 : nat`
- Quand la fonction est un nom, on peut se passer de parenthèses
  - Exemple: `Definition add2 := fun x => x + 2.` `Check add2 4.`
  - Le système répond `add2 4 : nat`

Les fonctions sont des valeurs

- Les fonctions peuvent retourner des fonctions comme résultat
  - C'est la représentation usuelle des fonctions à plusieurs arguments
  - Coq fournit une notation abrégée
- Exemple: une fonction avec deux entrées
  - `Check fun x => fun y => y + (x + 2).`
  - Le système répond
    `fun x y : nat => y + (x + 2) : nat -> nat -> nat`
  - Quand on l'applique à un seul argument cela retourne une fonction
  - La nouvelle fonction peut à son tour être appliquée à un autre argument
  - `Check (fun x y => y + (x + 2)) 5.`
  - Le système répond
    `(fun x y : nat => y + (x + 2)) 5 : nat -> nat`

Les fonctions peuvent être des arguments

- Les fonctions d'ordre supérieur
  - `Check fun f => f (f 0).`
  - Le système répond

    ```
    fun f : nat -> nat => f (f 0)
         : (nat -> nat) -> nat
    ```

  - Ceci indique que l'argument `f` est une fonction
- Exemple d'utilisation
  - ```
    Eval vm_compute in (fun f => f (f 0))
          ((fun x y => y + (x + 2)) 3).
    ```
  - Le système répond `10 : nat`
    ```

Alpha-conversion

- Dans une fonction le nom de la variable spéciale peut changer sans changer le sens de la fonction
  - `fun x => x + 2` et `fun y => y + 2` sont la même fonction,
  - Le changement du nom de variable est appelé *alpha-conversion*

Bêta-réduction

- Dans les calculs, la variable spéciale est remplacée partout par l'argument
  - `Eval vm_compute in (fun x f => f (f x)) 2.`
  - Le système répond

    ```
    fun f : nat -> nat => f (f 2)
        : (nat -> nat) -> nat
    ```
  - Cette étape élémentaire de calcul est appelée *Beta-réduction*

Variables liées, variables libres

- L'alpha-conversion et la beta-réduction sont reliées
  - Les variables touchées par les deux procédés sont des *variables liées*
  - l'occurrence entre `fun` et => est l'occurence *liante*
- Lorsqu'une variable dans une expression n'est pas liée, elle doit avoir été définie par ailleurs

Utilisation de variables locales

- La construction `let v := e in F`
- Cette construction permet d'écrire une seule fois `e`
- La variable `v` aura la même valeur et le même type que `e` et pourra être utilisée dans `F`
- `Eval compute in let v := 3 in v + v.`
- Le système répond
  `= 6 : nat`

Les types en Coq

Yves Bertot

Les types : outils principaux contre les fautes

- Les types sont des ensembles de données
- Ils permettent d'observer les programmes avec moins de détail
- Les erreurs les plus grossières se voient rapidement
- La vérification des types peut être faite sans exécuter les programmes

Les types

- Tout objet a un type,
- Toute fonction attend un argument d'un type précis,
- Même lorsque l'on ne donne pas de type à l'argument d'une fonction, le système en calcule un ou refuse,
- La commande `Check` permet de trouver le type d'une expression.

Quelques expressions et leur type

- `Check 1.`
- Le système répond:
  `1 : nat`
- `Check 1 + 2.`
- Le système répond:
  `1 + 2 : nat`
- `Check fun x => 1 + x.`
- Le système répond:
  `fun x : nat => 1 + x : nat -> nat`
- `Check (fun x => 1 + x) 2.`
- Le système répond:
  `(fun x : nat => 1 + x) 2 : nat`

Quelques erreurs de type

- `Check 1 + true.`
- Le système répond : `The term "true" has type "bool" while it is expected to have type "nat".`
- `Check fun x => x.`
- Le système répond :
  `Error: Cannot infer the type of x.`
- `Check 1 2.`
- Le système répond :
  `Error: Illegal application (Non-functional construction):`
  `The expression "1" of type "nat" cannot be applied to the term "2" : "nat"`

Règles de vérification des types

- Typage des applications de fonctions
  - Si une fonction `f` a le type A `->` B
  - Si l'on veut construire l'expression `f e`
  - Alors l'expression `e` doit avoir le type A.
- Typage des abstractions
  - Si l'on veut construire l'expression `fun x : t => M`
  - Dans l'expression `M`, `x` doit avoir le type `t`
- Si l'on n'écrit pas le type `t`, Coq essaie de deviner sa valeur

Polymorphisme

- Le comportement de certaines fonctions est commun à tous les types

```
let f := fun (A : Type) (x : A) => x in
    (f nat 3, f bool true)
```

- *Polymorphisme*: fréquent dans les langages comme ML, OCaml, Haskell,
- En Coq, le polymorphisme est explicite: l'argument de type est ajouté à la fonction,
- Notation liante: `forall`,
- Notation plus lourde, compensée par un mécanisme d'arguments implicites.

Arguments implicites

- La valeur de certains arguments peut être devinée quand on connaît le type d'autres arguments
  - Exemple `Definition id := fun (A:Type) (x:A) => x.`
  - `Check id _ true.`
  - Le système répond : `id bool true : bool`
  - Le système peut deviner `bool`
- `Implicit Arguments id.` permet d'utiliser cette capacité
  - On n'écrit plus que `id true`, l'autre argument est caché
  - On peut utiliser le préfixe `@` pour désactiver.
  - `Check @id bool.`
  - Le système répond : `id (A:=bool) : bool -> bool`

Ordre supérieur

- La notation `fun _ => _` permet de construire des fonctions,
- Des fonctions peuvent produire de nouvelles fonctions,
- Des fonctions peuvent prendre des fonctions en argument,
- Une fonction de premier ordre est une fonction qui prend en argument une donnée
  - nombres naturels, booléens, paires de données, etc.
- Une fonction de second ordre prend en argument une fonction du premier ordre,
- Une fonction d'ordre supérieur prend en argument une fonction d'ordre arbitraire,
- En Coq la distinction entre les ordres n'existe pas.

Notations pour l'ordre supérieur

- On écrit souvent des fonctions à un argument qui retournent une fonction à un argument,
- Les parenthèses dans les applications sont évitées (à droite),
- Les parenthèses dans le typage sont évitées,
- Les mots clefs `fun` ne sont pas répétés.

Les structures de données en Coq

Yves Bertot

Structures de données

- Pour programmer il faut pouvoir définir de nouvelles structures de données
- Il s'agit de regrouper des données ou de décrire des alternatives
- Des opérations sont fournies pour dégrouper et traiter les cas
- La récursion permet d'avoir dans le même type des objets de tailles différentes

Regrouper des données

- Un type prédéfini de couples, notation A `*` B
- Notation pour les données elles-mêmes `( . , . )`
  - `Check (3, fun x : nat => x + 3).`
  - Le système répond
    `(3, fun x => x + 3) : nat * nat -> nat`
- Besoin d'une construction pour observer l'intérieur d'un couple
- Observation par motif, syntaxe `match ... with ... end`
  - `Eval compute in`
    `match (1, 3) with (a, b) => a + b end.`
  - Le système répond
    `= 4 : nat`
- Le calcul fonctionne en comparant `(a, b)` avec la valeur observée `(1, 3)`
- `a` tombe en face de `1`
- `a` a la valeur `1` dans le calcul de `a + b`

Définir son propre type

- Un mot-clef : `Inductive`
  - `Inductive t1 : Type :=`
    `ct1 (x y : nat) (b : bool).`
  - Ceci définit deux nouveaux objets: `t1` et `ct1`
  - `t1` est un type
  - `ct1` est une fonction pour construire des éléments de `t1`
- observation par motif:
  - `Check fun d : t1 =>`
    `match d with ct1 u v w => u + v end.`
  - Le système répond
    `fun d : t1 =>`
    `match d with ct1 u v w => u + v end : t1 -> nat`
  - L'observation par motif permet de connaître le type de `u` et `v`

Structures de données avec variantes

- On peut avoir plusieurs cas de figure
- Par exemple un calcul peut retourner deux entiers ou une chaine de caractères
  - `Inductive t2 := ct2n (x y : nat) | ct2s (s : string).`
  - Toute donnée de `t2` est soit `ct2n n` où `n` est entier soit `tt ct2s s` où `s` est une chaîne de caractères
  - L'observation par motif devient un traitement par cas
-
```
Check fun c : t2 =>
        match c with
            ctn n => n + n
        |   cts s => 0
        end.
```
  - si `c` a été obtenu par `ctn` le premier calcul est effectué
  - si `c` a été obtenu par `cts` le deuxième calcul est effectué

Terminologie sur les structures de données

- Dans la définition inductive suivante:
  `Inductive t2 := ct2n (x y : nat) | ct2s (s : string).`
- Les nouvelles fonctions `ct2n` et `ct2s` sont appelées des *constructeurs*
- `x`, `y`, `s` sont des *champs*
- les types `nat` et `string` sont des types de champs

Structures de données récursives

- Les types de champs sont habituellement des types déjà définis
- Le type en cours de définition est aussi autorisé
- Exemple:
  `Inductive t3 := ct3r (s:string)(r:t3) | ct3e.`
- Ceci indique que
  - `ct3e` a le type `t3`
  - `ct3r "a" (ct3r "b" (ct3r "c" (ct3r "d" ct3e)))` aussi
- Tous les éléments de `t3` contiennent `ct3e` précédé d'un certain nombre d'utilisations de `ct3r`

Programmation récursive

- Pour observer le contenu d'un élément d'un type récursif
- Une construction qui va s'adapter aun nombre de fois où le type est utilisé de façon imbriquée
- Forme générale: `Fixpoint f (x : t) : t' := B.`
- Le type `t'` est le type de retour de la fonction
- La fonction `f` peut être réutilisée dans le corps de la fonction, `B`
  - Seulement sur une variable obtenue par traitement par cas sur `x`

Exemples de types récursifs utilisés dans Coq

- `Inductive nat : Set := O : nat | S : nat -> nat.`
- ```
  Inductive list (A : Type) : Type :=
      | nil : list A
      | cons : A -> list A -> list A.
  ```
- Check S (S (S O)).
- Le système répond
  `3 : nat`
- La fonction `S` correspond à l'operation "ajouter 1"
  ```

Exemples de programmation récursive

- La fonction qui additionne tous les nombres plus petits que `n`:

```
Fixpoint sum (n : nat) : nat :=
match n with
| O => O
| S p => p + sum p
end.
```

- La fonction qui calcule la suite de Fibonacci:

```
Fixpoint fib (n:nat) : nat :=
match n with
| O => O
| 1 => 1
| S (S q as p) => fib p + fib q
end.
```

- Notez l'utilisation de `as` pour respecter la contrainte de variable

Fonctions récursives sur d'autres types

- La fonction qui concatène toutes les chaines d'un objet de type `t3`

```
Fixpoint cat (x : t3) : string :=
match x with
| ct3e => ""
| ct3r s y => s ++ cat y
end.
```

Violation de la contrainte

- Message d'erreur lorsque la contrainte n'est pas respectée

```
Fixpoint fib' (n:nat) : nat :=
match n with
| O => O
| 1 => 1
| S (S q) => fib' (S q) + fib q
end.
```

- Le système répond

```
Recursive call to fib' has principal argument
equal to
"S q"
instead of one of the following variables:  n0 q.
```

Bibliothèque de base

Yves Bertot

Les nombres

- Par défaut les nombres utilisés en Coq sont des nombres naturels
- Il s'écrivent 0, 3, 47, 268
- Représentation interne naïve, `1 = S O`, `2 = S (S O)`
  - adaptée pour le raisonnement logique
- Pas de nombres négatifs dans ce type
- `Check 3.`
- Le système répond:
  `3 : nat`

Les opérations sur les nombres

- addition, multiplication, soustraction
- Notations infixes:
- `Check plus 1 2.`
- Le système répond:
  `1 + 2 :  nat`
- `Check mult 2 3.`
- Le système répond:
  `2 * 3 :  nat`
- soustraction totale : résultat irrégulier dans certains cas
- `Eval vm_compute in minus 2 3.`
- Le système répond:
  `= 0 :  nat`

Programmation par cas sur les entiers naturels

- Les entiers naturels sont de deux formes
  - 0
  - `S p` où `p` est un autre nombre naturel
- Il est possible de définir une valeur de façon différente suivant la forme d'une autre expression
- `match n with 0 => 1 | S p => 3 end`
- La valeur de cette expression est 1 si `n` est 0, 3 si `n` est non nul

Exemple de calcul avec traitement par cas

- ```
  Eval vm_compute in
          (fun n =>
                match n with 0 => 1 | S p => n + p
  end)
          (2+1).
  ```
- Le système répond:
  ```
  = 5 :  nat
  ```
  ```

Fonctions récursives sur les nombres naturels

- Il est possible de définir une fonction qui s'appelle elle-même
  - A condition que l'appel se fasse sur un prédécesseur de l'argument
  - Le prédecesseur doit être obtenu par traitement par cas
  - Cette contrainte garantit que le calcul s'arrête toujours
- Syntaxe particulière:

```
Fixpoint fact (n : nat) : nat :=
  match n with 0 => 1 | S p => n * fact p end.
```

Calcul d'une fonction récursive sur les entiers naturels

- Exemple de calcul:
  `Eval vm_compute in fact 6.`
- Le système répond:
  `= 720 : nat`
- La représentation naïve des entiers naturels empêche de calculer des grandes valeurs.

Les valeurs de vérité

- Un type `bool` contenant deux valeurs `true` et `false`
- Une construction `if ... then ... else ...` pour faire un calcul qui dépend d'un test
- Un certain nombre de fonction de test retournant une valeur booléenne
- `Search bool.`
- Le système répond

```
leb : nat -> nat -> bool
beq_nat : nat -> nat -> bool
andb : bool -> bool -> bool
orb : bool -> bool -> bool
```

- `beq_nat` est un test d'égalité pour les entiers naturels, `leb` un test de comparaison, `andb` calcule la conjonction, etc.

Mettre ensemble plusieurs données

- `(`e_1`,` e_2`)` est bien formé dès que e_1 et e_2 le sont
- si e_1 a le type t_1 et si e_2 a le type t_2
  alors `(`e_1`,` e_2`)` a le type t_1 `*` t_2.
- Lorsque l'on a une expression `e` de type `(`t_1 `*` t_2`)` on peut récupérer chacune des composantes grâce à l'observation de motif:
  `match e with (a, b) => F end` les deux variables `a` et `b` peuvent être utilisées dans l'expression `F`,
- les valeurs de `a` et `b` sont celles de la première composante de `e` et de la deuxième composante, respectivement.

Mettre ensemble un nombre quelconque de données

- pour tout type `t`, on peut utiliser le type `list t` pour mettre ensemble des données de type `t`
- La notation est `... :: ... :: ... :: nil`
- Le traitement par cas sur cette structure de donnée s'écrit aussi avec `match`, il permet d'isoler le premier élément d'une liste du reste, lorsque cet élément existe

  `match l with a::l' => F`$_1$ `| nil => F`$_2$ `end`

Programmation récursive sur les listes

- Dans le cas `...::...` une liste contient une sous-liste
- Si cette sous-liste est nommée par une variable, elle peut servir pour un appel récursif
- On utilise encore `Fixpoint` pour définir les fonctions récursives
- ```
  Fixpoint suml (l:  list nat) : nat :=
  match l with
  | nil => 0
  | a::tl => a + suml tl
  end.
  ```
- Cette fonction additionne toutes les valeurs d'une liste d'entiers.
  ```

Polymorphisme

Yves Bertot

Des fonctions générales

- Certains calculs sont indépendants du type des données
- Par exemple, si `A` est un type, `f` est une fonction à deux arguments dans `A` et à valeur dans `A` et si x_1, x_2, et x_3 alors le calcul `f` x_1 `(f` x_2 x_3`)` a toujours un sens
- On peut le décrire par une fonction de `A`, `f`, x_1, x_2, et x_3
- ```
  Definition op3 (A : Type) (f : A -> A -> A)
             (x1 x2 x3 : A) :=
    f x1 (f x2 x3).
  ```
- Cette fonction prend le type `A` en argument et va s'adapter à tous les types
- C'est une fonction *polymorphe*
  ```

Fonctions polymorphes : utilisation

- `Eval compute in op3 string append "a" "b" "c".`
- Le système répond
  `= "abc" :  string`
- `Eval compute in op3 nat plus 1 2 3.`
- Le système répond
  `= 6 :  nat`
- Le système peut deviner l'argument de type
  `Check op3 _ plus 1 2 3.`
- Le système répond
  `op3 nat plus 1 2 3 :  nat`

Le type des fonctions polymorphes

- Dans une fonction polymorphe, le nom de l'argument de type est réutilisé
- La notation `... -> ...` n'est plus adaptée
- Une nouvelle notation : `forall A : Type, ...`
- `Check op3.`
- Le système répond
  `op3 : forall A : Type, (A -> A -> A) -> A -> A -> A -> A.`
- En fixant les types, on retrouve des types habituels
- `Check op3 nat.`
- Le système répond
  `op3 nat : (nat -> nat -> nat) -> nat -> nat -> nat -> nat`

Arguments implicites

- Quand une fonction est polymorphe on peut adopter la convention de ne pas écrire l'argument de type
- L'argument manquant est deviné en observant le type de l'argument suivant
- La commande suivante met en place la convention pour une fonction
  `Implicit Arguments op3.`
- A partir de là on ne donne plus l'argument `A` à `op3`
- `Check op3 plus.`
- Le système répond
  `op3 plus :  nat -> nat -> nat -> nat`
- le préfixe `@` désactive le mécanisme d'arguments implicites
- `@op3 nat` est bien formé.

Structures de données polymorphes

- Le type des couples est polymorphe
- La notation `(..., ...)` cache une fonction `pair`
- Cette fonction a quatre arguments, deux types `A` et `B` et deux valeurs `x` et `y`
- Les arguments `A` et `B` sont implicites
- Le type de la fonction pair est
  `pair : forall A B : Type, A -> B -> A * B`
- Le type des couples est défini de la façon suivante:

```
Inductive prod (A B : Type) : Type :=
    pair : A -> B -> prod A B.
```

- `A * B` est une notation pour `prod A B`

Listes polymorphes

- La notation `...::...` cache une fonction `cons`
- `cons : forall A : Type, A -> list A -> list A`
- L'argument `A` est implicite
- Le type des listes est défini par

```
Inductive list (A : Type) : Type :=
| nil : list A
| cons : A -> list A -> list A.
```

- Un grand nombre des fonctions prédéfinies sur les listes sont polymorphes avec arguments implicites

Fonctions polymorphe sur les listes

- `length`, `nth`, `app`
- `Check app.`
- Le système répond
  `app:  forall A Type, list A -> list A -> list A`
- `Eval compute in app (1::2::nil) (3::nil).`
- Le système répond
  `= (1::2::3::nil)`