T'aider ? Hein ?

S. Chevillard, J. Duprat, F. Becker

6 février 2007

1 Utilisation de Coq

1.1 La commande `coqtop`

On peut utiliser la version installée de Coq en appelant la commande

```
coqtop
```

Le lancement de cette commande doit retourner la bannière suivante :

```
Welcome to Coq 8.0beta (Jan 2004)

Coq <
```

Pour ceux d'entre vous qui veulent disposer de leur propre version sur leur machine personnelle, je suggère de suivre les indications fournies sur la page Web du logiciel :

`http://coq.inria.fr/`

Je joins également à la fin de document des indications pour utiliser des aides diverses dans vos sessions futures.

1.2 Un peu de syntaxe

Cette section ne devrait normalement pas être nécessaire pour tous les élèves qui ont assisté au cours. Je l'inclus ici pour servir de rappel.

Dans les premières sessions nous n'utiliserons que la λ-abstraction et l'application de fonction du calcul des constructions. Le symbole λ n'est pas disponible sur les claviers usuels en France et Coq utilise une autre convention, l'expression $\lambda x : A.B$ est représentée par le texte suivant :

```
fun x:A => B
```

Pour les fonctions à plusieurs arguments on pourra utiliser plusieurs abréviations, basées sur des espaces pour séparer des arguments de même type et sur des parenthèses pour séparer des séquences d'arguments de type différent.

Par exemple, les expressions

"`fun (x:A) (y:A) (z:B) => y`" et "`fun (x y:A) (z:B) => y`" représentent la même fonction : la fonction qui prend deux arguments de type `A` et un argument de type `B` et retourne son deuxième argument.

L'application d'une fonction en argument est habituellement représentée par $f(x)$ en mathématique, mais Coq utilise une autre convention, plus proche de ocaml et de Lisp : l'application d'une fonction sera toujours représentée par la juxtaposition de la fonction et de l'argument, sans mettre de parenthèses autour de l'argument. Pour l'application des fonctions à plusieurs arguments, il existe également une abréviation qui consiste à faire disparaitre les paires de parenthèses à gauche.

Par exemple les expressions `((f a) b) c` et `f a b c` représentent la même valeur : la fonction `f` appliquée à `a`, `b` et `c`.

Cette convention se reporte de façon cohérente sur le parenthésage des expressions contenant des flêches (types de fonctions), mais cette cohérence s'exprime par le fait que l'on fait disparaitre les paires de parenthèses *à droite*.

1.3 Logique de l'implication

On se place dans le contexte où `A`, `B`, et `C` sont déclarées comme des variables de type `Type`. Construire des fonctions de type suivante (fournir leur écriture en syntaxe Coq).

1. `A -> A`
2. `A -> B -> A`
3. `A -> B -> B`
4. `A -> (A -> B) -> B`
5. `(A -> B) -> (B -> C) -> (A -> C)`
6. `(A -> B -> C) -> (A -> B) -> A -> C`

On pourra s'aider de la commande `coqtop` pour vérifier les résultats. Pour se placer dans le bon contexte, on devra d'abord envoyer la commande suivante :

```
Variables A B C : Type.
```

Ensuite pour tester une réponse on pourra utiliser la commande `Check`. Par exemple pour vérifier qu'une expression a bien le type

```
(A -> B -> C) -> B -> A -> C,
```

j'envoie la commande suivante :

```
Check (fun (f: A -> B -> C)(hb : B)(ha : A) => f ha hb).
```

La réponse de Coq est la suivante :

fun (f : A -> B -> C) (hb : B) (ha : A) => f ha hb
: (A -> B -> C) -> B -> A -> C

Ce qui confirme le type attendu.

1.4 Utilisation de types dépendants

On se place dans un contexte où A est un type, B et C sont des fonctions de A vers l'ensemble des types, f est une fonction qui associe à tout élément x de A un élément de $B(x)$ et g est une fonction qui associe à tout élément y de A et à tout élément z de $B(y)$ un élement de $C(y)$.

1. Peut-on construire une fonction h qui associe à tout élement x de A un élément de $C(x)$?
2. Combien d'arguments la fonction g prend-elle ?
3. On peut progressivement abstraire la fonction h par rapport aux différents éléments qui entrent dans sa composition. Quels sont les types que l'on obtient ?
4. Décrire la session Coq qui permet de répondre à ces différentes questions.

1.5 Un exemple difficile

Construire une fonction dont le type représente la formule logique suivante :

$$\forall A, B : Prop.((((A \Rightarrow B) \Rightarrow A) \Rightarrow A) \Rightarrow B) \Rightarrow B$$

2 Du lambda-calcul et du système F

2.1 Curry-Howard

- Rappeler ce qu'est l'isomorphisme de Curry-Howard.
- Dans cet isomorphisme, à quoi correspond le modus ponens ? Toujours dans ce cadre, en quoi consiste la vérification d'une preuve ?

2.2 λ-calcul simplement typé

On se place maintenant dans le cadre du λ-calcul simplement typé. Soit τ le type de base. On pose $\mathbb{N}_0 = \tau \to \tau$, $\mathbb{N}_{k+1} = \mathbb{N}_k \to \mathbb{N}_k$.

- Qu'est-ce que $\mathbb{N}_1$? Voyez-vous comment l'identifier à $\mathbb{N}$?
- Peut-on écrire un λ-terme de type $\mathbb{N}_1 \to \mathbb{N}_1 \to \mathbb{N}_1$ qui représente l'addition ?
- Et la multiplication ?
- Et l'exponentiation : $(n, p) \mapsto n^p$?

2.3 Système F

On se place maintenant dans le sytème F. Voyez-vous une manière de sortir de l'impasse précédente ?

- Constater que tout se transpose bien dans ce formalisme : 0, successeur, addition, multiplication.
- Que dire de l'exponentiation ?

2.4 Balivernes

Puisqu'on a su faire les entiers, on peut sans doute faire à peu près tout. En fait, on peut coder toute la logique (mais Jean Duprat se fera un plaisir de vous le montrer au prochain cours). Un exemple : le faux peut se représenter par $\forall X.X$. D'après vous, pourquoi est-ce une bonne idée ?

- Comment est typée la fonction suivante en Caml : `let rec f x = f x` ? Cette fonction termine-t-elle habituellement ?
- Montrez que $\forall X.\forall Y.X \rightarrow Y$ est équivalent à faux.

2.5 Question bonus :

En vous inspirant des entiers de Church, sauriez-vous concevoir un type pour représenter un type somme quelconque de Caml ?