T'aider ? Hein ?

S. Chevillard, D. Pous

5 février 2008

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 8.0pl3 (Jan 2006)

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/
```

Vous pouvez utiliser `coqide` qui est une interface graphique à Coq.

1.2 Un peu de syntaxe

Dans les premières sessions nous n'utiliserons que la λ-abstraction et l'application de fonction du calcul des constructions. En Coq, l'expression $\lambda(x : A).x$ est représentée par

```
fun (x:A) => x
```

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 types différents.

```
fun (x:A) (y:A) (z:B) => y
fun (x y : A) (z:B) => y
fun (_ y:A) (_:B) => y
```

sont trois définitions du même objet.

Les règles de parenthésage des types, de même que les règles d'écriture pour l'application d'une fonction à ses arguments sont identiques en Coq et en OCaml :

```
f a b c == (((f a) b) c).
```

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`. Pour cela, on tape les commandes suivantes à l'invite de Coq :

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

On aurait pu écrire aussi :

```
Variable A B C: Type.
```

L'usage veut qu'on utilise `Variable` lorsqu'une seule variable est déclarée et `Variables` lorsqu'on en déclare plusieurs.

Notez l'usage du point, qui a pour effet d'inviter Coq à interpréter les instructions. C'est l'équivalent du double point-virgule d'OCaml.

Vous pouvez définir un objet à l'aide du mot clé `Definition` :

```
Definition f := (fun x:A => x).
Definition f (x:A) := x.
```

Comme OCaml, Coq infère le type automatiquement la plupart du temps. Il est tout de même possible de forcer le typage de la façon suivante :

```
Definition f (x:A) : A := x.
```

Remarquez que Coq râle si vous vous trompez :

```
Definition f (x:A) : B := x.
```

Question : Construire des fonctions ayant les types suivants :

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`

Utilisez la commande `Check` pour vérifier le type de chacune de vos fonctions.

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 $C(y)$.

Questions : Écrire la suite d'instructions Coq qui crée ce contexte.

1. Peut-on construire une fonction h qui associe à tout élement x de A un élément de $C(x)$?
2. 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 ?

1.5 Un exemple difficile

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

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

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 ? Donnez deux termes différents de type `(A->C) -> (B->C) -> A -> B -> C`. On a donc deux types pour un même terme : à quoi cela correspond-il dans l'isomorphisme de Curry-Howard ? Que vous inspire le terme suivant :
  `fun (Fa : A -> C) (Fb : B -> C) (a:A) (b:B) => Fa ((fun x y => y) b a).`

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 $\forall X.\forall Y.X \to Y$ 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 ?