Fondements des systèmes de preuves – TP n° 8

Les sortes de Coq

Le système de types de Coq est construit sur les sortes de base `Prop` (la sorte des propositions), `Set` (la sorte des types de données de base), et la *hiérarchie cumulative* d'univers $\texttt{Type}_i$ dont les indices $i \geq 1$ sont gérés de manière implicite par le système. Les axiomes sous-jacents sont :

$$\texttt{Prop} : \texttt{Type}_1, \qquad \texttt{Set} : \texttt{Type}_1, \qquad \texttt{Type}_i : \texttt{Type}_{i+1} \qquad (i \geq 1)\,.$$

(On a aussi les inclusions $\texttt{Prop} \subset \texttt{Type}_1$, $\texttt{Set} \subset \texttt{Type}_1$ et $\texttt{Type}_i \subset \texttt{Type}_{i+1}$.) La sorte `Prop` est imprédicative, la sorte `Set` prédicative,[1] et tous les univers $\texttt{Type}_i$ sont prédicatifs.

Exercice 1 (`Type` / `Type`)

1. Définir une constante `MonType` égale à `Type`. Quel est le type de `MonType` ?
2. Définir une fonction identité de type `MonType` → `MonType`, et appliquer cette fonction à `MonType`. Que se passe-t-il ? Pourquoi ?

Exercice 2 (Booléens dans `Prop`, `Set` et `Type`) À côté des booléens dans `Set` (type `bool`), on définit les booléens dans `Prop` (`boolP`) et dans `Type` (`boolT`) en posant :

```
Inductive boolP : Prop :=
| trueP  : boolP
| falseP : boolP.
```

```
Inductive boolT : Type :=
| trueT  : boolT
| falseT : boolT.
```

Définir les fonctions de conversion

- `bool_to_boolP : bool -> boolP` et `boolP_to_bool : boolP -> bool`.
- `bool_to_boolT : bool -> boolT` et `boolT_to_bool : boolT -> bool`.
- `boolP_to_boolT : boolP -> boolT` et `boolT_to_boolP : boolT -> boolP`.

Que se passe-t-il ?

L'existence de deux sortes (presque) jumelles `Prop` et `Set` sert à distinguer au niveau de l'extraction les types dont les objets ont un contenu algorithmique réel (`Set`, `Type`) des types dont les objets sont des preuves (`Prop`) qui n'ont pas à être conservées lors de l'extraction. C'est pour cette raison qu'il est impossible de discriminer deux preuves d'une même proposition (telles que `trueP` et `falseP`), puisque celles-ci sont vouées à disparaître à l'extraction.

La co-existence des sortes `Prop` et `Set` nécessite de dupliquer les constructeurs de sommes et de sommes dépendantes suivant le niveau où l'on désire placer les arguments et/ou le résultat :

Les sommes dépendantes		
Nom et type	**Syntaxe**	**Type extrait**
`ex (T : Set) (T -> Prop) : Prop`	`exists x : T, P x`	`unit`
`sig (T : Set) (T -> Prop) : Set`	`{x : T | P x}`	`T`
`sigS (T : Set) (T -> Set) : Set`	`{x : T & P x}`	`T * P`

[1] Depuis la version 8.0 du système seulement. Voir l'option : `-set-impredicative`.

Les sommes directes			
Nom et type		**Syntaxe**	**Type extrait**
`or`	`(A : Prop) (B : Prop) : Prop`	`A \/ B`	`unit`
`sum`	`(A : Set) (B : Set) : Set`	`A + B`	`(A, B) sum`
`sumbool`	`(A : Prop) (B : Prop) : Set`	`{A} + {B}`	`bool`
`sumor`	`(A : Set) (B : Prop) : Set`	`A + {B}`	`A option`

Exercice 3 (Extraction)

1. Démontrer : `forall x : nat, {y : nat | x = 2 * y \/ x = 2 * y + 1}`.
   Où la démonstration bloque-t-elle ?
2. Démontrer : `forall x : nat, {y : nat & {x = 2 * y} + {x = 2 * y + 1}}`.
   Extraire le programme correspondant en Caml. Quel est le type de ce programme ?
3. Résoudre la question 1 en utilisant le résultat de la question 2, et extraire.

Exercice 4 (*Proof-irrelevance*) On considère les deux propositions suivants :

- La *proof-irrelevance*,[2] qui exprime que toute proposition a au plus une preuve :
  `forall A : Prop, forall x y : A, x = y`
- L'*extensionnalité propositionnelle*, qui dit que deux propositions équivalentes sont égales :
  `forall A B : Prop, (A <-> B) -> A = B`

Le but de l'exercice est de montrer que l'axiome d'extensionnalité propositionnelle implique la *proof-irrelevance*. Dans la suite de l'exercice, on suppose la propriété d'extensionnalité propositionnelle (posée en axiome).

Un type A est un *rétracte* d'un type B s'il existe un couple de fonctions $f : A \to B$ (le *plongement*) et $g : B \to A$ (la *rétraction*) telles que $g \circ f = \mathrm{id}_A$. La relation de rétraction sur les types propositionnels est définie par :

```
Definition retr (A B : Prop) :=
  exists f : A -> B, exists g : B -> A, forall x : A, g (f x) = x.
```

1. Montrer que la relation définie ci-dessus est réflexive. En déduire à l'aide de l'axiome d'extensionnalité propositionnelle que pour toute proposition A habitée (par au moins une preuve), la proposition $A \to A$ est un rétracte de A.

Intuitivement, un type A tel que $A \to A$ est un rétracte de A est un modèle du λ-calcul pur : l'injection `lam` : $(A \to A) \to A$ correspond à l'abstraction, la rétraction `app` : $A \to (A \to A)$ à l'application, et l'égalité `app` $\circ$ `lam` $= \mathrm{id}_{A \to A}$ à la β-réduction.

2. En utilisant l'intuition ci-dessus, montrer que si $A \to A$ est un rétracte de A, alors toute fonction de A dans A admet un point fixe. (*Indication* : On montre ce résultat en construisant un combinateur de point fixe sur A.)
3. En déduire que la négation sur le type propositionnel `boolP` (cf exercice 0) admet un point fixe, d'où l'égalité `trueP` = `falseP`.
4. En déduire que toutes les preuves d'une proposition A (quelconque) sont égales.
5. Que se passe-t-il si on remplace `Prop` par `Set` dans tout ce qui précède ?
6. Refaire l'exercice dans le calcul des constructions pur (i.e. sans types inductifs), en utilisant les codages imprédicatifs de l'égalité, de la quantification existentielle et des booléens.

[2] L'expression *proof-irrelevance* peut se traduire par « indiscernabilité des preuves ».