Vérification (HAI603I)

Licence Informatique
Département Informatique
Faculté des Sciences de Montpellier
Université de Montpellier

TD/TP N°6 : Prouver = Programmer

Exercice 1 (Logique implicative minimale)

Démontrer les propositions suivantes en utilisant les règles de la déduction naturelle vues en cours. Puis, transformer l'arbre de preuve en un arbre de typage pour le λ-calcul simplement typé. Dans chaque cas, vous indiquerez explicitement quel est le λ-terme représentant la preuve de la proposition.

1. $A \Rightarrow B \Rightarrow A$
2. $(A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C$

Exercice 2 (Ajout de la conjonction)

Dans cet exercice, on va ajouter la conjonction « $\wedge$ » à la logique implicative minimale et voir ce que cela rajoute côté λ-calcul afin de préserver l'isomorphisme d'Howard. Côté règles de preuve, l'ajout de la conjonction rajoute les règles suivantes :

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge_I$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge_{E1} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge_{E2}$$

Dans le λ-calcul simplement typé, afin de capturer ce nouveau connecteur, nous devons modifier le langage des termes et le langage des types. Concernant les termes, nous devons rajouter les couples, ainsi que les projecteurs comme suit :

— Si t_1 et t_2 sont des termes, alors (t_1, t_2) est un terme.

— Si t est un terme, alors fst t et snd t sont des termes.

Côté langage des types, nous devons rajouter le produit cartésien de la manière suivante :

— Si τ_1 et τ_2 sont des types, alors $\tau_1 \times \tau_2$ est un type.

Les règles de typage de ces nouveaux termes sont définies comme suit :

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1, t_2) : \tau_1 \times \tau_2} \mathsf{Tup}$$

$$\frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \text{fst } t : \tau_1} \mathsf{Fst} \qquad \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \text{snd } t : \tau_2} \mathsf{Snd}$$

Sachant que dans l'isomorphisme de Curry-Howard, nous souhaitons faire correspondre les règles de preuve $\wedge_I$, $\wedge_{E1}$, et $\wedge_{E2}$, aux règles de typage Tup, Fst, et Snd respectivement, répondre aux questions suivantes :

1. Écrire les fonctions de correspondance Φ et φ pour ce nouveau connecteur (le « $\wedge$ ») et les nouvelles règles de preuve correspondantes.
2. Donner les λ-termes correspondants aux preuves de $A \Rightarrow B \Rightarrow A \wedge B$ et $A \wedge B \Rightarrow A$.

Exercice 3 (Extraction de la fonction factorielle)

L'objectif de cet exercice est de montrer comment il est possible d'extraire des fonctions à partir de preuves en Coq en utilisant l'exemple de la fonction factorielle en particulier.

Pour ce faire, nous allons utiliser le type `sig`, qui est équivalent au « il existe » mais adapté à l'extraction. La notation pour utiliser ce type est la suivante : `{x : A | P x}`, qui signifie « il existe un x de type A vérifiant $P(x)$ ».

1. Spécifier le comportement de la fonction factorielle en utilisant une relation inductive. Nous appellerons cette relation `is_fact` et elle prendra deux arguments (l'entier dont on veut calculer la factorielle et le résultat).

2. Démontrer le lemme suivant :

   ```
   Lemma fact : forall n : nat, { v : nat | is_fact n v }.
   ```

   Une fois la preuve terminée, vous devrez sauvegardez la preuve avec la commande `Defined` (qui permet de préparer l'extraction).

3. Extraire la fonction factorielle de la preuve du lemme `fact` effectuée précédemment. Pour ce faire, il faut importer la bibliothèque dédiée à l'extraction, puis utiliser la commande d'extraction (par défaut, l'extraction produit du code OCaml) comme suit :

   ```
   Require Extraction.
   Extraction fact.
   ```

   Cette commande n'extraie que la fonction correspondant au lemme `fact`. Pour extraire toutes les dépendances (notamment le type des entiers naturels), il faut utiliser la commande suivante :

   ```
   Recursive Extraction fact.
   ```

   Utiliser cette commande et exécuter le code produit dans un toplevel OCaml sur plusieurs exemples (attention, les entiers utilisés correspondent aux entiers extraits de Coq, à savoir les entiers de Peano).

Exercice 4 (Extraction de l'égalité sur les entiers naturels)

Cet exercice est similaire au précédent. Nous nous proposons d'extraire une fonction qui teste l'égalité entre deux entiers naturels.

Pour ce faire, nous allons utiliser le type `sumbool` (que nous avons déjà vu au TD/TP précédent), qui est équivalent au « ou » mais adapté à l'extraction. On rappelle que la notation pour utiliser ce type est la suivante : `{A} + {B}`, qui signifie « A ou B ».

1. Démontrer le lemme suivant :

   ```
   Lemma eq_nat : forall n m : nat, {n = m} + {n <> m}.
   ```

   Pour faire cette preuve, on doit faire une induction structurelle sur `n` et sur `m`. Cette double induction peut se faire soit en enchaînant les inductions, soit en utilisant la tactique `(double induction n m)`. À noter également qu'il existe une tactique en Coq qui démontre les lemmes de décidabilité de l'égalité automatiquement, il s'agit de la tactique `decide equality` (vous pouvez la tester sur notre exemple). Comme dans l'exercice précédent, Une fois la preuve terminée, vous devrez sauvegardez la preuve avec la commande `Defined` afin de préparer l'extraction.

2. Extraire la fonction d'égalité sur les entiers naturels et la tester sur plusieurs exemples en OCaml. Comme la fonction en Coq retourne un objet de type `sumbool` (et non un booléen), il est à noter que la fonction OCaml retournera des objets de ce type extrait, à savoir les valeurs `Left` (correspondant à `true`) et `Right` (correspondant à `false`).

Exercice 5 (Extraction de l'inversion d'une liste)

Cet exercice est similaire aux deux exercices précédents. Nous nous proposons d'extraire une fonction qui inverse l'ordre des éléments d'une liste d'entiers naturels.

1. Spécifier le comportement de la fonction qui inverse l'ordre des éléments d'une liste d'entiers naturels en utilisant une relation inductive. Nous appellerons cette relation `is_rev` et elle prendra deux arguments (la liste d'entiers naturels dont on veut inverser l'ordre des éléments et le résultat).

2. Démontrer le lemme suivant :

   ```
   Lemma rev : forall l : list nat, {l' : list nat | is_rev l l'}.
   ```

   Comme dans les deux exercices précédents, une fois la preuve terminée, vous devrez sauvegardez la preuve avec la commande `Defined` afin de préparer l'extraction.

3. Extraire la fonction qui inverse l'ordre des éléments d'une liste d'entiers naturels et la tester sur plusieurs exemples en OCaml.