Logique constructive et types dépendants

Preuves assistées par ordinateur – TP 1 – 31 janvier 2025

Année scolaire 2024–2025

Exercice 1 : Un peu de théorie constructive des ensembles

On se donne deux ensembles `A` et `B`, qu'on définit une fois pour toutes les définitions qui vont suivre.

```
Parameters A B : Set.
```

1. Définir une propriété `injective: (A -> B) -> Prop` sur les fonctions `A -> B` telle qu'on ait `injective f` lorsque `f` est injective (*c.-à.-d.*, pour tous x et y, si $fx = fy$, alors $x = y$).
2. Montrer que `f` est injective si et seulement si `f` est simplifiable à gauche (*c.-à.-d.*, pour toutes fonctions g et g', si $f \circ g = f \circ g'$, alors $g = g'$).
3. Définir une propriété `surjective: (A -> B) -> Prop` sur les fonctions `A -> B` telle qu'on ait `surjective f` lorsque `f` est surjective (*c.-à.-d.*, pour tout y, il existe x, tel que $fx = y$).
4. En supposant le tiers exclu, montrer que `f` est surjective si et seulement si `f` est simplifiable à droite (*c.-à.-d.*, pour toutes fonctions g et g', si $g \circ f = g' \circ f$, alors $g = g'$).
   Erratum : On supposera aussi l'axiome d'extentionnalité propositionnelle :
   `forall p q : Prop, (p <-> q) -> p = q`
   On peut noter que la conjonction de l'axiome d'extentionnalité propositionnelle et du tiers exclu est équivalente à l'axiome de dégénérescence propositionnelle (ou complétude propositionnelle) : `forall p : Prop, p = True \/ p = False`. Ces axiomes sont définis dans le module `ClassicalFacts` de la bibliothèque standard.
   Indications : Le tiers exclu permet de supposer par l'absurde
   `~ (exists x : A, f x = y)`
   On pourra poser les définitions suivantes :
   ```
   set (g := fun _: B => False).
   set (g' := fun y' => y = y').
   ```
   On peut alors montrer facilement que `g y <> g' y`, et la contradiction vient du fait qu'on peut aussi montrer `g y = g' y` avec l'extentionnalité propositionnelle.
5. Peut-on se passer du tiers exclu ?
   Indications : On peut en effet se passer du tiers exclu, mais pas de l'axiome d'extentionnalité propositionnelle. On pourra poser les définitions suivantes :
   ```
   set (g := fun (y: B) => True).
   set (g' := fun (y: B) => exists x, f x = y).
   ```

La tactique `replace` e `with` e' permet de remplacer dans le but les occurrences de e par e' en ajoutant le but supplémentaire $e' = e$. On peut s'en servir pour remplacer le but `exists x : A, f x = y` par `True`, ce qui nous ramènera à prouver
`True = (exists x : A, f x = y)`
qui est convertible en `g x = g' y` (on peut utiliser la tactique `change` pour passer d'un but convertible à un autre).

Exercice 2 : Lemme d'affaiblissement

On se donne un ensemble de variables.

```
Parameter var : Set.
```

1. Définir un inductif `lambda` pour représenter les termes du λ-calcul (on demande trois constructeurs, pour les variables, les abstractions et les applications).
   Indication : voici une définition possible pour le premier constructeur

```
Inductive lambda :=
| Var (x: var)
| ...
```

   ce qui est un raccourci d'écriture pour

```
Inductive lambda :=
| Var : forall x: var, lambda (* ou, de manière équivalente, var -> lambda *)
| ...
```

2. Définir un inductif `type` pour représenter les types (on demande seulement deux constructeurs : les types atomiques, qui seront nommés par des variables, et l'implication).
   Indication : lorsque plusieurs paramètres ont le même type, on peut les écrire sous la même annotation de type

```
Inductive type :=
| ...
| Arrow (t u: type).
```

3. On représentera un contexte par une valeur de type `list (var * type)`. Définir un inductif `derivation` pour représenter les dérivations de typage en logique naturelle : cet inductif aura trois indices, pour le contexte, le λ-terme et son type. On aura un constructeur pour chacune des trois règles suivantes : la règle axiome et les règles d'introduction et d'élimination de l'implication.
   Indications : on pourra importer le module `List` de la bibliothèque standard avec la commande vernaculaire `Require Import List.`, et utiliser le prédicat d'appartenance `In` qui y est défini pour la règle d'axiome.

```
Inductive derivation: context -> lambda -> type -> Prop :=
| Ax: forall Gamma, forall x t, In (x, t) Gamma -> derivation Gamma (Var x) t
| ...
```

4. Montrer le lemme d'affaiblissement sur ce fragment : pour tous Γ, x et t, si $\Gamma \vdash x : t$, alors pour tout Γ', si $\Gamma \subseteq \Gamma'$, alors $\Gamma' \vdash x : t$.
   Indication : on pourra utiliser `incl` du module `List` pour exprimer $\Gamma \subseteq \Gamma'$. Pour voir les théorèmes déjà définis pour `In` et `incl`, on pourra utiliser les commandes `Search In` et `Search incl.`

5. On pourra itérativement élargir le fragment aux autres règles de la logique naturelle.

Exercice 3 : Diagonale d'une matrice carrée

On se donne le type des listes de longueur fixée.

```
Inductive vect {A}: nat -> Type :=
| nil: vect O
| cons: forall {n}, A -> vect n -> vect (S n).
```

Définir une fonction `diag: forall A n, @vect (@vect A n) n -> @vect A n` qui extrait la diagonale d'une matrice carrée.

Indications : on commencera par définir les fonctions auxiliaires suivantes.

— `head : forall {A} {n}, @vect A (S n) -> A`
— `tail : forall {A} {n}, @vect A (S n) -> @vect A n`
— `map : forall {A} {B} {n} (f: A -> B), @vect A n -> @vect B n`

En observant `Print head`, `Print tail` et `Print map`, comprendre comment Coq/Rocq a pu éliminer les cas de filtrage impossibles.

On fera d'abord un filtrage sur `n` et on utilisera le principe de coupure commutative (ou *convoy pattern*) : lorsqu'on filtre sur une valeur `x` et que `x` et/ou l'un des indices apparaissant dans le type de `x` apparaît également dans le type d'autres variables du contexte, on peut η-expanser ces variables autour du `match` pour que leur type soit réécrit en fonction des branches. Par exemple, si on a un contexte avec `n: nat` et `mat: @vect (@vect A (S n)) (S n)`, on peut abstraire `mat` dans le `match` pour que les occurrences de `n` dans le type de `map` soient remplacées par la valeur prise par `n: nat` dans chacune des branches.

```
match n as n0 return forall mat: @vect (@vect A n0) n0, ... with
 | O => fun mat: @vect (@vect A O) O => ...
 | S m => fun mat: @vect (@vect A (S m)) (S m) => ...
end mat.
```

Noter que le prédicat de retour, `as n0 return forall mat: @vect (@vect A n0) n0, ...`, peut le plus souvent être inféré par Coq/Rocq.

En particulier, si on a dans le contexte `mat: @vect (@vect A (S m)) (S m)`, abstraire le témoin d'égalité `eq_refl : n = n` permet de récupérer dans la branche `cons` un témoin d'égalité `m = m'`, où `m'` est le paramètre de longueur porté par `cons`.

```
match mat with
| @cons _ m' hd tl => fun e : m = m' => ...
end eq_refl
```

La même technique permet également d'utiliser un tel témoin d'égalité pour réécrire `m'` en `m` dans le type de `tl : @vect (@vect A m) m'`.

```
match e in _ = m0 return forall tl : @vect (@vect A (S m)) m0, @vect A (S m) with
| eq_refl => fun (tl : @vect (@vect A (S m)) m) => ...
end (tl : @vect (@vect A (S m)) m')
```

Ici encore, les annotations de type sur `tl` et le prédicat de retour sont superflus.