Rapport TP3

Valeran MAYTIE

1 Datatypes and Simple Inductive Proofs

En Isabelle, on peut définir des listes inversées en s'inspirant des listes usuelles. Pour cela, on crée 2 constructeurs : `Nil` et `Snoc` (Figure-1)

```
datatype 'a List =
  Nil
| Snoc "'a List" 'a
```

FIGURE 1 – Liste inversées

On peut donc par exemple créé la liste : `Snoc (Snoc Nil (1::nat)) 2::nat` $\equiv$ `[1; 2]`

Ensuite, on peut écrire des fonctions qui vont faire des opérations sur ces listes. On va écrire 3 fonctions : `filter`, `map` et `concat` (Figure-2) :

```
fun filter
where
  "filter f Nil        = Nil"
| "filter f (Snoc l a) =
    (if f a
     then  Snoc (filter f l) a
     else filter f l)"

fun map
where
  "map f Nil        = Nil"
| "map f (Snoc l a) = Snoc (map f l) (f a)"

fun concat
  where
  "concat S Nil = S"
| "concat S (Snoc l a) = Snoc (concat S l) a"
```

FIGURE 2 – Fonction sur les listes

Une fois que les fonctions sont créées, on peut commencer à spécifier les fonctions à l'aide de formule logique. Dans le TP, on veut prouver deux formules :

— $\mathit{filter}\ p\ (\mathit{filter}\ q\ S) = \mathit{filter}\ (\lambda x.p\ x \wedge q\ x)\ S$

— $\mathit{map}\ f\ (\mathit{concat}\ R\ S) = \mathit{concat}\ (\mathit{map}\ f\ R)(\mathit{map}\ f\ S)$

Pour prouver ces deux lemmes, on va utiliser les théorèmes de récurrence créés quand on déclare un type à l'aide de `datatype`. On peut les lister grâce à `find_theorems name:List name:TP3 name:induct`. Après avoir trouvé le bon théorème, on peut l'appliquer grâce à `apply (rule_tac List="?" in List.induc)`. Le point d'interrogation doit être remplacé par la liste sur laquelle on veut faire la récurrence. Ici, on veut la faire sur la liste S, car les fonctions font la récurrence sur celle-ci.

Enfin après avoir appliqué le théorème, il faut simplifier les expressions ce qui fini la preuve. On peut raccourcir la preuve en utilisant la tactique `induct_tac "S"`.

2 Inductive sets - Inductive Proofs

Dans cette section, on cherche à définir un prédicat inductif pour définir les ensembles contenant uniquement les nombre pairs (Figure-3).

```
inductive_set Even :: "int set"
where
  a: "0 ∈ Even"
| b: "n ∈ Even ⇒ (n + 2) ∈ Even"
| c: "n ∈ Even ⇒ (n - 2) ∈ Even"
```

FIGURE 3 – Ensemble des nombres pairs

Les notations a, b et c permettent de donner des noms aux lemmes associés. On peut donc maintenant prouver assez facilement que $2 \in \mathit{Even}$ (Figure-4.

```
lemma "2 ∈ Even"
  using Even.a Even.b by force
```

FIGURE 4 – Ensemble des nombres pairs

Malheureusement, montrer que $1 \notin \mathit{Even}$ est bien plus difficile. Pour faire cella, il faut d'abord montrer : $x \in \mathit{Even} \Rightarrow \exists k.\ x = 2 \times k$ et ensuite, on peut appliquer ce lemme et montrer qu'il n'existe pas de k telle que $1 = 2 \times k$.

3 Modeling Exercise

Dans cette partie, on va chercher à modéliser le λ-calcul dans Isabelle. Pour cella, on commence à définir le type des λ-termes (Figure-5)

```
datatype const =
    Nat nat
  | Bool bool
```

```
datatype terms =
  Var string
| Const const
| App terms terms
| Lambda string terms
```

FIGURE 5 – λ-termes avec constante

Ensuite, on veut modéliser la règle de typage vue dans le premier cours. Donc, on définit les types possibles (Figure-6)

```
datatype types =
  TVar string
| TNat
| TBool
| Arrow types types (infixr "→" 200)
```

FIGURE 6 – Définitino des types

On peut définir des notations assez simplement (plus facilement quand Coq) à l'aide du mot clé `infixr/l`. Le type fonction sera donc écrit avec une flèche dans les spécifications.

Maintenant, il faut définir la fonction `instantiate`. Elle va remplacer un type variable par un autre type. Elle s'écrit assez simplement par récurrence sur les types (Figure-7).

```
fun instantiate :: "types ⇒ string ⇒ types ⇒ types"
where
  "instantiate (t1 → t2) s t = (instantiate t1 s t) → (instantiate t2 s t)"
| "instantiate TNat s t = TNat"
| "instantiate TBool s t = TBool"
| "instantiate (TVar vs) s t = (if vs = s then t else (TVar vs))"
```

FIGURE 7 – Fonction `instantiate`

Enfin, on peut modéliser les règle d'inférence vue dans le premier cours à l'aide d'un prédicat inductif (Figure-8)

```
inductive typing :: "(const ⇀ types) ⇒ (string ⇀ types) ⇒ terms ⇒ types ⇒ bool"
    ("\_, \_ ⊢ \_ : \_" [50, 50, 50] 50)
  where
    Var   : "Γ x = Some T ⇒ Σ, Γ ⊢ Var x : T"
  | Const : "Σ c = Some C ⇒ Σ, Γ ⊢ Const c : instantiate C x T"
  | Abs   : "Σ, Γ (v ↦ T) ⊢ t : U ⇒ Σ, Γ ⊢ Abs v t : (T → U)"
  | App   : "Σ, Γ ⊢ t1 : (T2 → T1) ⇒ Σ, Γ ⊢ t2 : T2 ⇒ Σ, Γ ⊢ (App t1 t2) : T1"
```

FIGURE 8 – Fonction `instantiate`

Pour représenter les environnements, on utilise les `Map` de Isabelle qui sont représenter dans le type par une fonction partiel (⇀).

Pour rendre les règles plus lisibles, on va défini la même notation qui est utilisée sur papier ce qui donne un prédicat inductif assez joli et visuel.