Master Informatique FIIL — 2016–17
Preuves Interactives et Applications — 5 octobre 2016
http://www.lri.fr/~paulin/PreuvesInteractives

Examen Session 2 - 24 juin 2016

L'examen dure 3 heures. L'énoncé est composé de **??** pages.
Le barème est indicatif. Toutes les réponses devront être clairement justifiées.
Documents autorisés : copie des transparents de cours, sujets et corrigés du TD disponibles sur le site à l'exclusion de tout autre document.

Le tableau résumant les règles de la déduction naturelle est donné à la fin du sujet.
Inscrivez votre nom sur chaque copie et numérotez-la.
Cacheter toutes les copies, recopier le numéro d'anonymat sur les intercalaires !

Exercice 1 (Questions de cours, 4 points)

Dire si les affirmations suivantes sont vraies ou fausses dans chacun des systèmes Isabelle/-HOL et Coq. On attend donc deux réponses à chaque fois. Une bonne réponse compte 1 point, une mauvaise réponse fait perdre $\frac{1}{2}$ point sur cet exercice. On ne demande pas de justification.

1. On peut traduire chaque terme qui représente une fonction en un programme Ocaml.
2. On peut formaliser (sans axiome) la fonction qui décide pour une codage d'une machine Turing si l'exécution s'arrête pour toutes les entrées.
3. On peut prouver dans ces systèmes tout ce qui est vrai, leur logique est complète.
4. On peut exprimer des types qui dépendent d'objets comme les entiers (exemple le type des listes de longueur n).
5. Le système est toujours capable d'inférer un type pour les objets quand un tel type existe.

Exercice 2 (λ-calcul simplement typé, 2 points)

On rappelle que la constante $\forall$ a pour type $(\alpha \to \texttt{Prop}) \to \texttt{Prop}$, la constante $\_=\_$ a pour type $\alpha \to \alpha \to \texttt{Prop}$, la constante $\_+\_$ a pour type $\texttt{nat} \to \texttt{nat} \to \texttt{nat}$ et la constante 0 a pour type $\texttt{nat}$. On utilise une notation infixe $x + y$ (resp. $x = y$) pour le résultat de l'application de la constante $\_+\_$ (resp. $\_=\_$) aux termes x et y.

Dire si les expressions suivantes sont bien typées dans le λ-calcul simplement typé et si oui donner le type le plus général, si non expliquer pourquoi.

1. $\lambda f.\lambda x.f\,(f\,x)$
2. $\lambda f.\lambda x.f\,(x\,f)$
3. $\lambda P.\forall(\lambda x.\forall(\lambda y.(P\,x\,y)))$
4. $\lambda f.(f\,\forall(\lambda x.((x + x) = 0)))$

Exercice 3 (Preuves, 4 points)

Trouver et corriger au moins 5 coquilles ou erreurs dans les définitions et preuves du script suivant. Pour trouver les erreurs, il peut-être utile de dessiner les peuves à la main.

Cet exercice peut être réalisé au choix de l'étudiant en Coq ou en Isabelle. Par conséquent, il faut choisir **l'un des deux scripts** ci-dessous :

- **Script Isabelle**

```
lemma and_sym: "A ⟶B ⟶B ∧A"
apply(rule impI)
apply(rule conjI)
apply(erule conjE)
apply(assumption)
done

lemma ex_distr: "(∃x. (A ⟶P(x))) = (A ⟶(∃x. P(x)))"
apply(rule iffI)
 apply(erule exE)
  apply(rule impI)
 apply(simp)
apply(rule exI)
apply(case_tac A, simp_all)
done

(* la declaration suivante est incorrecte, corriger *)
datatype 'a seq = Empty | Seq 'a 'a seq


fun filter:: "('a ⇒bool) ⇒'a seq ⇒'a seq"
where     "filter P Empty = Empty"
|            "filter P (Seq x xs) = (if P x then Seq x (filter P xs) else filter P xs)"


fun conc :: "'a seq ⇒'a seq ⇒'a seq"
where
  "conc Empty ys = ys"
| "conc (Seq x xs) ys = Seq x (conc xs ys)"

lemma filter_conc [simp]: "filter P (conc xs ys) = (conc (filter P xs)(filter P ys))"
  apply(induct ys)
  apply(simp_all)
  done

lemma filter_filter [simp]: "filter P (filter Q xs) = filter (λx. Q x ∧P x) xs"
  by (induct xs) simp
```

Rappel : les règles `iffI`, `mp` et `spec` ont la forme :

```
"(P ⟹Q) ⟹(Q ⟹P) ⟹P = Q"
"(P ⟶Q) ⟹P ⟹Q"
"(∀x. P x) ⟹P x"
```

- **Script Coq** :

```
Lemma and_sym (A B : Prop) : A → B → B /\ A.
intro H.
split.
destruct H.
assumption.
Qed.

Lemma ex_distr X (A : Prop) (P : X → Prop) :
    (∃ x, (A → P x)) ↔ (A → (∃ x, P x)).
split; intro H.
+destruct H as (x,HP).
intro Ha.
∃ x.
+ destruct H as (x,HP).
∃ x.
assumption.
Qed.

Set Implicit Arguments.

(* la declaration suivante est incorrecte, corriger *)
Inductive seq A := Empty : seq A | Seq : A → seq → seq A.

Fixpoint conc A (xs ys: seq A) : seq A :=
  match xs with Empty ⇒ ys
              | Seq x xs ⇒ Seq x (conc xs ys)
  end.

Fixpoint filter A (P : A → bool) (xs : seq A) : seq A :=
  match xs with Empty ⇒ Empty A
              | Seq x xs ⇒ if (P x) then Seq x (filter P xs) else filter P
  end.


Lemma filter_conc A (P : A → bool) xs ys
   : filter P (conc xs ys) = conc (filter P xs) (filter P ys).
induction ys.
simpl; auto.
Qed.

Lemma filter_filter A (P Q : A → bool) xs
   : filter P (filter Q xs) = filter (fun x ⇒ andb (P x) (Q x)) xs.
induction xs.
+simpl; auto.
+simpl; auto.
Qed.
```

On rappelle que `A <-> B` est défini comme `A -> B /\ B -> A` et donc que la tactique `split` engendre deux sous-buts `A -> B` et `B -> A`.

Exercice 4 (Modélisation, 10 points)
Cet exercice peut être réalisé au choix en utilisant la syntaxe de Coq ou celle d'Isabelle.

Un arbre AVL est un type particulier d'arbre binaire de recherche avec une certaine proprieté, dite d'être AVL-balancé (d'après les auteurs en 1962 Adelson-Velski-Landis). Cette propriété d'équilibre relatif garantit des temps d'accès, d'insertion ou de retrait logarithmiques.

Les données stockées dans l'arbre sont dans un type A qui est muni d'une opération de comparaison `compare` dont le résultat `compare` $x\,y$ appartient à l'ensemble $\{\texttt{Lt}, \texttt{Eq}, \texttt{Gt}\}$. Cet ensemble est défini comme un type inductif à trois valeurs constantes. La fonction `compare` vérifie les propriétés suivantes :

- $(\texttt{compare}\,x\,y = \texttt{Eq}) \Leftrightarrow (x = y)$
- $(\texttt{compare}\,x\,y = \texttt{Lt}) \Leftrightarrow (\texttt{compare}\,y\,x = \texttt{Gt})$
- $(\texttt{compare}\,x\,y = \texttt{Lt}) \wedge (\texttt{compare}\,y\,z = \texttt{Lt}) \Rightarrow (\texttt{compare}\,x\,z = \texttt{Lt})$

On dira que x est plus petit que y si $\texttt{compare}\,x\,y = \texttt{Lt}$.

Le type des arbres (ordinaires) est donné par la déclaration suivante en Coq

```
Inductive tree A := leaf : tree A | node : A → tree A → tree A → tree A
```

ou en Isabelle/HOL

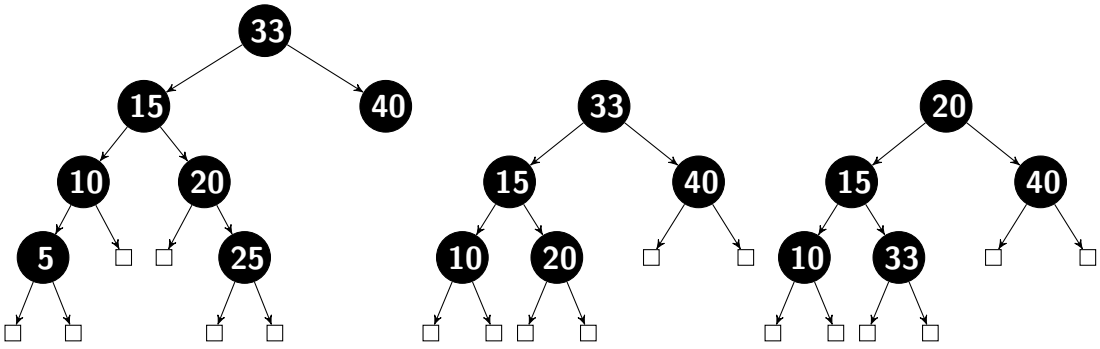
```
datatype 'a tree = leaf | node 'a "'a tree" "'a tree"
```

On rappelle qu'un arbre binaire de recherche a la propriété que pour chaque nœud interne $(\texttt{node}\,n\,l\,r)$, les valeurs stockées dans le sous-arbre gauche l sont plus petites que n qui est lui-même plus petit que toutes les valeurs stockées dans le sous-arbre droit r.

On peut definir la "hauteur" d'un arbre binaire comme la longueur maximale d'un chemin de la racine à une feuille. La proprieté caractéristique pour un arbre AVL est d'être AVL balancé qui signifie que pour chaque nœud interne de l'arbres, les longueurs des sous-arbres gauche et droite diffèrent au plus d'un.

Questions :

1. Dire si les arbres suivants sont bien des arbres AVL sur les entiers. Les nœuds internes sont représentés par des ronds et les feuilles par des carrés.



2. Définir une fonction `height` qui calcule l'hauteur d'un arbre binaire.
3. Définir un type `bal` à trois valeurs qui contient les éléments `Left`, `Right`, `Bal`.

4. Définir une fonction `form` sur les arbres qui renvoie `Left` si le sous-arbre gauche de la racine est strictement plus haut que le sous-arbre droite, `Bal` si les deux sous-arbre ont même hauteur et `Right` si le sous-arbre gauche de la racine est strictement moins haut que le sous-arbre droite.
5. Définir de manière récursive une fonction `isin` qui étant donnés une valeur a dans A et un arbre t d'éléments de A qui vérifie de plus la propriété d'être un arbre binaire de recherche, renvoie vrai si et seulement si la valeur a apparaît dans un nœud de t.
6. Definir de manière récursive une fonction `is_avl` qui décide si un arbre de type `tree` satisfait toutes les propriétes pour être un arbre binaire AVL.
7. On veut definir une fonction `insert` qui étant donnés une valeur a et un arbre AVL t renvoie un arbre AVL qui contient tous les éléments de t plus la valeur a.
   (a) Préciser le type de la fonction `insert`.
   (b) Ecrire en utilisant les fonctions définies précédemment des formules logiques qui spécifient le comportement de la fonction `insert`.
   (c) Définir de manière récursive la fonction `insert` (la solution n'a pas besoin d'être efficace).

Rappel des règles logiques

Hypothèse	si $A \in \Gamma$ alors $\Gamma \vdash A$	
Connecteur	Introduction	Élimination
$\top$	$\Gamma \vdash \top$	
$\bot$		$\dfrac{\Gamma, \neg C \vdash \bot}{\Gamma \vdash C}$
$\neg$	$\dfrac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}$	$\dfrac{\Gamma \vdash \neg A \quad \Gamma \vdash A}{\Gamma \vdash C}$
$\wedge$	$\dfrac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$	$\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \quad \dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$
$\vee$	$\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \quad \dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B}$	$\dfrac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$
$\Rightarrow$	$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}$	$\dfrac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$
$\forall$	$\dfrac{\Gamma \vdash P \quad x \notin \mathsf{FV}(\Gamma)}{\Gamma \vdash \forall x, P}$	$\dfrac{\Gamma \vdash \forall x, P}{\Gamma \vdash P[x \leftarrow t]}$
$\exists$	$\dfrac{\Gamma \vdash P[x \leftarrow t]}{\Gamma \vdash \exists x, P}$	$\dfrac{\Gamma \vdash \exists x, P \quad \Gamma, P \vdash C \quad x \notin \mathsf{FV}(\Gamma, C)}{\Gamma \vdash C}$
$=$	$\Gamma \vdash \forall x, x = x$	$\dfrac{\Gamma \vdash t = u \quad \Gamma \vdash P[x \leftarrow t]}{\Gamma \vdash P[x \leftarrow u]}$