

Preuves assistées par ordinateur**Examen du 9 mai 2021 – durée : 3h – avec typos corrigées**

Documents autorisés

*Le barème ci-dessous est indicatif et est susceptible d'être modifié.***Exercice 1 (≈ 6 points) – Dédution naturelle**

1. Les séquents suivants sont-ils dérivables en déduction naturelle ? Si oui, en donner des dérivations en forme normale (au choix : soit en style arbre de formules, soit en style arbre de séquents, soit sous la forme d'un λ -terme de preuve). Si non, donner un contre-modèle (au choix : avec un modèle booléen qui rend le séquent faux ou en analysant les formes normales possibles).

(a) $(A \vee B) \Rightarrow C \vdash (A \Rightarrow C) \wedge (B \Rightarrow C)$

(b) $(A \Rightarrow C) \wedge (B \Rightarrow C) \vdash (A \vee B) \Rightarrow C$

(c) $(\exists x A(x)) \Rightarrow B \vdash \forall x (A(x) \Rightarrow B)$

(d) $\forall x (A(x) \Rightarrow B) \vdash (\exists x A(x)) \Rightarrow B$

(e) $(A \vee B) \wedge C \vdash A \vee (B \wedge C)$

2. On considère la preuve suivante de $\exists x x \geq 0 \Rightarrow \exists x x \geq 0$ représentée en λ -calcul :

$$H_1 := \lambda a. \text{dest } a \text{ as } (x, b) \text{ in } (x + 1, Ax_1 x b)$$

où Ax_1 est une preuve de $\forall x (x \geq 0 \Rightarrow x + 1 \geq 0)$.

On considère la preuve suivante de $\exists x x \geq 0$, elle aussi formulée en λ -calcul :

$$H_2 := (3, Ax_2)$$

où Ax_2 est un axiome prouvant $3 \geq 0$.

On considère la preuve suivante de $(\exists x x \geq 0 \Rightarrow \exists x x \geq 0) \Rightarrow (\exists x x \geq 0 \Rightarrow \exists x x \geq 0)$:

$$H_3 := \lambda f. \lambda x. f(fx)$$

Donner la forme normale de la preuve $H := H_3 H_1 H_2$ de $\exists x (x \geq 0)$ (on pourra si nécessaire faire référence à Ax_1 , Ax_2). En particulier, quel est la valeur du témoin justifiant l'introduction du \exists ?

Exercice 2 (≈ 3 pts) : Iteration, analyse de cas et récursion

On considère le type suivant d'arbres à branchement dénombrable, défini en Coq par :

```
Inductive tree :=
| Leaf : nat -> tree
| Node : (nat -> tree) -> tree
```

On suppose ce type équipé d'un schéma d'itération :

```
tree_iter : forall B:Type, (nat -> B) ->
  ((nat -> B) -> B) ->
  tree -> B
```

1. Indiquer comment prouver à partir de `tree_iter` un schéma de récursion `tree_rec` de type :

```
tree_rec : forall B:Type, (nat -> B) ->
  ((nat -> tree) -> (nat -> B) -> B) ->
  tree -> B
```

On donnera la preuve sous la forme d'un λ -terme, en syntaxe informelle ou syntaxe Coq. Notez qu'on pourra éventuellement s'aider de types standard tels que le produit ou la somme, et/ou s'inspirer de l'« astuce de Kleene ».

2. Indiquer comment prouver à partir de `tree_rec` le schéma d'analyse de cas `tree_case` de type suivant :

```
tree_case : forall B:Type, (nat -> B) ->
  ((nat -> tree) -> B) ->
  tree -> B
```

Là aussi, on donnera la preuve sous la forme d'un λ -terme, en syntaxe informelle ou syntaxe Coq.

Exercice 3 (≈ 16 pts) : Prouvabilité et validité en logique propositionnelle classique

Notez que certaines questions peuvent être traitées indépendamment.

A – Interprétation booléenne des formules propositionnelles On considère le type suivant `form` de formules propositionnelles, paramétré par un ensemble d'atomes `I` et défini en Coq par :

```
Inductive form (I:Type) :=
| Atom : I -> form I          (* un atome de la forme X_i pour i∈I *)
| True  : form I              (* proposition tout le temps vraie *)
| False : form I              (* proposition tout le temps fausse *)
| Or    : form I -> form I -> form I  (* disjonction *)
| And   : form I -> form I -> form I  (* conjonction *)
| Imp   : form I -> form I -> form I  (* implication *)
```

On appelle formule close une formule sans atomes, c'est-à-dire de type :

Definition `form_closed := form Empty_set.`

où `Empty_set` est le type vide (avec éliminateur `Empty_set_rect` dont le type est celui-ci : `forall A, Empty_set -> A`).

On veut associer à toute formule propositionnelle une valeur de vérité `true` ou `false` dans le type `bool`. Par exemple, la valeur de vérité de la formule `True` est `true`, la valeur de `And f1 f2` est la conjonction binaire de la valeur de la vérité de `e1` et de la valeur de vérité de `e2`, etc. En particulier, pour l'implication, on prendra une interprétation classique, c'est-à-dire que `Imp f1 f2` est vraie si `f2` est vraie ou si `f1` est fausse.

1. Dans un premier temps, on veut calculer la valeur de vérité d'une formule propositionnelle close, c'est-à-dire sans atomes, c'est-à-dire d'une formule de type `form_closed`. Écrire ainsi en Coq une fonction `truth_closed : form_closed -> bool` qui calcule une telle valeur de vérité.
2. Indiquer en particulier le résultat de l'évaluation de
`truth_closed (Imp (And False True) False)`
et de
`truth_closed (Or (And True False) False)`
3. Dans un deuxième temps, on veut calculer la valeur de vérité d'une formule propositionnelle avec atomes indexés par des valeurs dans un certain type `I`. Pour cela, on doit associer à chaque atome `Atom i` pour `i:I` une valeur de vérité. On représentera une telle affectation de valeurs de vérité aux atomes par une fonction `σ : I -> bool` telle que `σ i` sera la valeur donnée à `Atom i`.
Écrire ainsi une fonction
`truth : forall {I:Type}, (I -> bool) -> form I -> bool`
qui calcule la valeur de vérité d'une formula avec atomes (les accolades autour de `I` indique que l'argument `I` de `truth` plus loin sera laissé implicite).
4. Indiquer en particulier le résultat de l'évaluation de
`truth sigma (Imp (And (Atom 0) (Atom 1)) (Atom 2))`
d'abord quand `sigma` est l'affectation qui associe `true` à tous les atomes sauf l'atome `Atom 1`, c'est-à-dire l'affectation définie par
`Definition sigma n := match n with 1 => false | _ => true end.`
puis quand `sigma` est l'affectation qui associe `true` à tous les atomes sauf l'atome `Atom 0`, c'est-à-dire l'affectation définie par
`Definition sigma n := match n with 0 => false | _ => true end.`
5. On dit qu'une formule est valide quand elle vraie pour toutes les affectations possibles de valeurs de vérité aux atomes. Par exemple, `Imp (Atom 0) (Atom 0)` est valide puisque vraie quelque soit la valeur de vérité de l'atome `Atom 0`.
Pour commencer, remarquez que la validité d'une formule close coïncide avec sa vérité puisqu'il n'y a pas d'atomes à instancier.
Ensuite, on considère les formules avec une seule variable propositionnelle, ce que l'on peut représenter par le type
`Definition form_one_atom := form unit.`
puisque `unit` est le type avec un seul habitant `tt` et que `form_one_atom` ne pourra ainsi contenir que le seul atome `Atom tt`.
Écrire une fonction `valid_one_atom : form_one_atom -> Prop` qui exprime qu'une formule à un seul atome est vraie pour tous les affectations possibles de valeur de vérité à cet atome (il y a deux affectations possibles).
6. Montrer que la formule `Or F (Imp F False)` (tiers-exclu) est valide pour toute formule `F`.
7. On définit le type `context` des contextes de formules avec atomes indexés sur `I` comme étant le type des listes de formules :
`Definition context (I:Type) := list (form I).`

On dit qu'un contexte est vrai pour une affectation quand toutes les formules du contexte sont valides. Écrire une fonction

```
truth_context : forall {I:Type}, (I -> bool) -> context I -> bool
```

qui calcule la vérité d'un contexte pour une affectation donnée de valeurs de vérité aux atomes.

B – Équivalence entre validité et prouvabilité avec tiers-exclu

8. On définit le type inductif suivant qui représente en déduction naturelle les preuves classiques d'une logique propositionnelle dont les atomes sont indexés par un ensemble I.

```
Inductive provable I:Type : list (form I) -> form I -> Prop :=
| Ax : forall F Γ, In F Γ -> provable Γ F
| TrueI : forall Γ, provable Γ True
| FalseE : forall Γ F, provable Γ False -> provable Γ F
| ImpI : forall Γ F G, provable (F::Γ) G -> provable Γ (Imp F G)
| ImpE : forall Γ F G, provable Γ (Imp F G) -> provable Γ F -> provable Γ G
| OrI1 : forall Γ F G, provable Γ F -> provable Γ (Or F G)
| OrI2 : forall Γ F G, provable Γ G -> provable Γ (Or F G)
| OrE : forall Γ F G H, provable Γ (Or F G) ->
    provable (F::Γ) H -> provable (G::Γ) H -> provable Γ H
| AndI : forall Γ F G, provable Γ F -> provable Γ G -> provable Γ (And F G)
| AndI1 : forall Γ F G, provable Γ (And F G) -> provable Γ F
| AndI2 : forall Γ F G, provable Γ (And F G) -> provable Γ G
| ExcludedMiddle : forall Γ F, provable Γ (Or F (Imp F False)).
```

où $::$ est le constructeur de listes et où $\text{In } F \Gamma$ est un prédicat qui dit que F est présent dans la liste Γ .

Montrer que toute formule avec au plus un atome qui est prouvable est valide, c'est-à-dire, prouver le théorème suivant (appelé le théorème de « correction » pour la logique propositionnelle classique) :

```
Theorem soundness_one_atom :
  forall (Γ:context unit) (F:form unit),
    provable Γ F ->
    forall σ:unit->bool, truth_context σ Γ = true -> truth σ F = true.
```

On pourra procéder par induction et se contenter de quelques cas, tel que le cas `Imp` et le cas `ExcludedMiddle`.

9. Donner une idée de comment on pourrait prouver la réciproque (appelée théorème de « complétude ») :

```
Theorem completeness_one_atom :
  forall (Γ:context unit) (F:form unit),
    (forall σ:unit->bool, truth_context σ Γ = true -> truth σ F = true) ->
    provable Γ F.
```

10. En déduire un théorème clé de la logique : la correspondance entre prouvabilité avec tiers-exclu et validité vis à vis de l'interprétation booléenne des formules (on pourra donner une preuve en mots, ou avec un terme de preuve) :

```
Theorem soundness_completeness :
  forall (F:form unit), valid_one_atom F <-> provable nil F.
```

où `nil` est le contexte vide.