Université de Paris M1 Informatique

Preuves assistées par ordinateur

Correction de la version améliorée de l'examen du 3 mai 2021 – durée : 3h

Documents autorisés

Le barème ci-dessous est indicatif et est susceptible d'être modifié.

Exercice 1 (≈ 6 points) – Déduction naturelle

1. Les énoncés a, b, c et e sont dérivables. On donne leur preuve sous la forme d'un lambda-terme (en utilisant la syntaxe du poly < proofs and programs » mais il aurait aussi été possible de donner un arbre de séquents.

   Pour chaque séquent, on nomme c l'unique hypothèse.

   (a) $(\neg A) \vee B \ \vdash \ A \Rightarrow B$

   Preuve : $\lambda a.\mathsf{case}\ c\ \mathsf{of}\ [\iota_1(d) \to \mathsf{efq}\,(d\,a) | \iota_2(b) \to b]$

   À titre d'exemple, on donne aussi la dérivation non annotée :

$$\dfrac{\dfrac{\dfrac{}{(\neg A) \vee B, A \ \vdash \ (\neg A) \vee B}\,Ax \qquad \dfrac{\dfrac{\dfrac{}{\neg A, A \ \vdash \ \neg A}\,Ax \quad \dfrac{}{\neg A, A \ \vdash \ A}\,Ax}{\neg A, A \ \vdash \ \bot}}{\neg A, A \ \vdash \ B}\,\bot_E \qquad \Rightarrow_E \quad \dfrac{}{B, A \ \vdash \ B}\,Ax}{(\neg A) \vee B, A \ \vdash \ B}\,\vee_R}{(\neg A) \vee B \ \vdash \ A \Rightarrow B}\,\Rightarrow_I$$

   ainsi que la dérivation annotée :

$$\dfrac{\dfrac{\dfrac{}{c : (\neg A) \vee B, A \ \vdash \ c : (\neg A) \vee B}\,Ax \qquad \dfrac{\dfrac{\dfrac{}{d : \neg A, a : A \ \vdash \ d : \neg A}\,Ax \quad \dfrac{}{d : \neg A, a : A \ \vdash \ a : A}\,Ax}{d : \neg A, a : A \ \vdash \ d\,a : \bot}}{d : \neg A, a : A \ \vdash \ \mathsf{efq}\,(d\,a) : B}\,\bot_E \qquad \Rightarrow_E \quad \dfrac{}{b : B, a : A \ \vdash \ b : B}\,Ax}{c : (\neg A) \vee B, a : A \ \vdash \ \mathsf{case}\ c\ \mathsf{of}\ [\iota_1(d) \to \mathsf{efq}\,(d\,a) | \iota_2(b) \to b] : B}\,\vee_R}{c : (\neg A) \vee B \ \vdash \ \lambda a.\mathsf{case}\ c\ \mathsf{of}\ [\iota_1(d) \to \mathsf{efq}\,(d\,a) | \iota_2(b) \to b] : A \Rightarrow B}\,\Rightarrow_I$$

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

   Preuve : $\mathsf{case}\ c\ \mathsf{of}\ [\iota_1(a) \to (\iota_1(\pi_1(a)), \pi_2(a)) | \iota_2(b) \to (\iota_2(\pi_1(b)), \pi_2(b))]$

   (c) $\forall x\,(A(x) \wedge B(x)) \ \vdash \ (\forall x\,A(x)) \wedge (\forall x\,B(x))$

   Preuve : $(\lambda x.\pi_1(c\,x), \lambda x.\pi_2(c\,x))$

   (d) $(\exists x\,A(x)) \wedge (\exists x\,B(x)) \ \vdash \ \exists x\,(A(x) \wedge B(x))$

   Ce n'est pas prouvable. En effet, de l'hypothèse on obtient un x tel que $A(x)$ et un x' tel que $B(x')$ mais rien ne dit qu'on pourra forcer ce x et ce x' à être les mêmes.

   (e) $\exists x\,(A \Rightarrow B(x)) \ \vdash \ A \Rightarrow (\exists x\,B(x))$

   Preuve : $\lambda a.\mathsf{dest}\ c\ \mathsf{as}\ (x, f)\ \mathsf{in}\ (x, f\,a)$

2. On considère la preuve suivante de $\forall n\,\forall m\,\exists r\,(r \geq m \wedge r \geq n)$ représentée en λ-calcul :

$$H_1 := \lambda n.\,\lambda m.\,((n+1) * (m+2), (Ax_1\,n\,m, Ax_2\,n\,m))$$

où Ax_1 et Ax_2 sont des axiomes donnés.
On considère la preuve suivante de $\exists p\,(p \geq m \wedge p \geq n) \Rightarrow \exists q\,(q \geq 0)$, elle aussi formulée en λ-calcul :

$$H_2 := \lambda H.\,\texttt{match}\ H\ \texttt{with}\ (p,(x,y))\ \Rightarrow\ (p+1, Ax_3\,(p+1))\ \texttt{end}$$

où Ax_3 est un axiome.
On considère la preuve suivante de $\exists q\,(q \geq 0)$:

$$H := H_2\,(H_1\,3\,4)$$

Donner la forme normale de la preuve H de $\exists q\,(q \geq 0)$ (on pourra si nécessaire faire référence à Ax_1, Ax_2 et Ax_3). En particulier, quel est la valeur du témoin justifiant l'introduction du $\exists$?

On applique la β-réduction sur la preuve et obtient :

$$\begin{array}{lll} H & \rightarrow & \texttt{match}\ (H_1\,3\,4)\ \texttt{with}\ (p,(x,y))\ \Rightarrow\ (p+1, Ax_3\,(p+1))\ \texttt{end} \\ & \rightarrow^* & \texttt{match}\ ((3+1)*(4+2), (Ax_1\,3\,4, Ax_2\,3\,4))\ \texttt{with}\ (p,(x,y))\ \Rightarrow\ (p+1, Ax_3\,(p+1))\ \texttt{end} \\ & \rightarrow^* & ((3+1)*(4+2)+1), Ax_3((3+1)*(4+2)+1))) \\ & \rightarrow^* & (25, Ax_3 25) \end{array}$$

Notez qu'on a réduit le rédex de tête (= appel par nom) mais qu'on aurait pu aussi l'évaluer dans un ordre différent pour factoriser plus de calcul (par exemple avec l'appel par valeur).

En particulier, le témoin de l'existentiel est 25.

Exercice 2 ($\approx$ 4 pts) : Schémas inductifs pour un type d'expressions arithmétiques

On considère le type `expr` défini en Coq par :

```
Inductive expr :=
| Cst : nat -> expr               (* Constante numérique:  n       *)
| Add : expr -> expr -> expr      (* Expression somme:     e1 + e2 *)
| Mul : expr -> expr -> expr.     (* Expression produit:   e1 * e2 *)
```

1. Indiquer le schéma de récursion canoniquement associé au type `expr` pour construire des objets de type `B`.

```
rec_expr : (nat -> B) ->
           (expr -> B -> expr -> B -> B) ->
           (expr -> B -> expr -> B -> B) ->
           expr -> B
```

2. Indiquer le schéma d'induction canoniquement associé au type `expr` pour prouver des propriétés de la forme `P e` pour `P` un prédicat et `e` une expression.

```
ind_expr : (forall n:nat, P (Cst n)) ->
           (forall e1:expr, P e1 -> forall e2:expr, P e2 -> P (Add e1 e2)) ->
           (forall e1:expr, P e1 -> forall e2:expr, P e2 -> P (Mul e1 e2)) ->
           forall e:expr, P e
```

3. Indiquer le schéma d'analyse de cas dépendant canoniquement associé au type `expr` pour prouver (sans appel récursif) des propriétés de la forme `T e` pour `T` un type dépendant et `e` une expression.

```
case_expr : (forall n:nat, P (Cst n)) ->
            (forall e1:expr, forall e2:expr, P (Add e1 e2)) ->
            (forall e1:expr, forall e2:expr, P (Mul e1 e2)) ->
            forall e:expr, P e
```

4. Indiquer une définition alternative de `expr` comme type du Système F dont les habitants en forme normale sont les codages de Church des habitants de `expr`.

```
forall X, (nat -> X) -> (X -> X -> X) -> (X -> X -> X) -> X
```

Exercice 3 (≈ 10 pts) : Évaluation d'expressions arithmétiques en Coq

A – Expressions arithmétiques On considère une nouvelle fois le type `expr` défini dans l'exercice 2.

1. Définir en Coq une fonction récursive `value : expr -> nat` calculant la valeur d'une expression.

```
Fixpoint value e :=
  match e with
  | Cst n => n
  | Add e1 e2 => value e1 + value e2
  | Mul e1 e2 => value e1 * value e2
  end.
```

B – Machine à environnement On considère une « machine à environnement » permettant d'évaluer des expressions. Les états de la machine, notés $\langle e \,|\, k \rangle$ sont des paires d'un code e de type `expr` et d'une continuation k de type `cont`. Le type `cont` est défini comme suit :

```
Inductive cont :=
| End : cont                     (* Fin du calcul *)
| AddL : expr -> cont -> cont    (* dit d'évaluer le 2e arg d'une addition *)
| AddR : nat -> cont -> cont     (* dit de finaliser le calcul d'une addition *)
| MulL : expr -> cont -> cont    (* dit d'évaluer le 2e arg d'une multiplication *)
| MulR : nat -> cont -> cont.    (* dit de finaliser le calcul d'une multiplication *)
```

La machine obéit aux règles de réduction suivantes :

instructions disant comment évaluer le code

$\langle$`Add` e_1 e_2 $\,|\,$ $k\rangle$	$\longrightarrow$	$\langle e_1 \,|\,$ `AddL` e_2 $k\rangle$	évaluer e_1 avant de continuer par l'addition à e_2
$\langle$`Mul` e_1 e_2 $\,|\,$ $k\rangle$	$\longrightarrow$	$\langle e_1 \,|\,$ `MulL` e_2 $k\rangle$	évaluer e_1 avant de continuer par la multiplication avec e_2

instructions disant comment continuer quand le code est déjà évalué

$\langle$`Cst` $n \,|\,$ `AddL` e $k\rangle$	$\longrightarrow$	$\langle e \,|\,$ `AddR` n $k\rangle$	évaluer e avant de continuer par l'addition à n
$\langle$`Cst` $n \,|\,$ `AddR` m $k\rangle$	$\longrightarrow$	$\langle$`Cst` $(m+n) \,|\, k\rangle$	réaliser l'addition de n et m
$\langle$`Cst` $n \,|\,$ `MulL` e $k\rangle$	$\longrightarrow$	$\langle e \,|\,$ `MulR` n $k\rangle$	évaluer e avant de continuer par la multiplication avec n
$\langle$`Cst` $n \,|\,$ `MulR` m $k\rangle$	$\longrightarrow$	$\langle$`Cst` $(m*n) \,|\, k\rangle$	réaliser la multiplication de n et m

Pour évaluer une expression e, la machine commence dans l'état initial $\langle e \,|\,$ `End`$\rangle$. Elle termine dans le seul type d'état pour lequel aucune règle ne s'applique, à savoir $\langle$`Cst` $n \,|\,$ `End`$\rangle$.

2. On considère le type « union disjointe » défini par :

```
Inductive sum (A:Type) (B:Type) :=
| inl : A -> sum A B
| inr : B -> sum A B.
```

Écrire en Coq une fonction `reduce : expr * cont -> sum nat (expr * cont)` qui prend en argument un état et si c'est un état final $\langle$`Cst` n | `End`$\rangle$ retourne le résultat n et, sinon, applique une étape de réduction et retourne le nouvel état.

```
Definition reduce ek :=
  match ek with
  | (Cst n, End) => inl n
  | (Add e1 e2, k) => inr (e1, AddL e2 k)
  | (Mul e1 e2, k) => inr (e1, MulL e2 k)
  | (Cst n, AddL e k) => inr (e, AddR n k)
  | (Cst n, AddR m k) => inr (Cst(m+n), k)
  | (Cst n, MulL e k) => inr (e, MulR n k)
  | (Cst n, MulR m k) => inr (Cst(m*n), k)
  end.
```

3. On considère le type de trace d'exécution suivant :

```
CoInductive trace :=
| result : nat -> trace          (* La machine a atteint un état final <Cst n|End> *)
| reduction : (expr * cont) -> trace -> trace.  (* La machine continue de réduire *)
```

Écrire une fonction `exec : expr -> trace` qui produit la trace de l'évaluation de la machine à environnement, c'est-à-dire la suite des étapes produites par `reduce` (on pourra si on le souhaite utiliser des fonctions auxiliaires).

```
CoFixpoint exec' ek :=
  match reduce ek with
  | inl n => result n
  | inr ek' => reduction ek' (exec' ek')
  end.

Definition exec e := exec' (e, End).
```

4. On considère le type « option » défini par :

```
Inductive option (A:Type) :=
| Some : A -> option A
| None : option A.
```

Écrire en Coq une fonction `compute : expr -> nat -> option nat` qui prend en argument une expression e et un entier n représentant un nombre d'étapes de calcul à faire, et qui exécute le processus suivant : initialiser la machine avec l'expression e, réduire n fois, renvoyer `Some` p si la machine a atteint un état final $\langle$`Cst` p | `End`$\rangle$ et renvoyer `None` sinon. On pourra si on le souhaite réutiliser `exec` et des fonctions auxiliaires.

Une possibilité est la suivante :

```
Fixpoint tail trace n :=
  match trace with
  | result n => Some n
```

```
  | reduction _ trace' =>
    match n with
    | 0 => None
    | S n' => tail trace' n'
    end
  end.

Definition compute e n := tail (exec e) n.
```

C – Sémantique

5. Définir un prédicat inductif `multireduce : expr * cont -> expr * cont -> Prop` qui prend en argument deux états et qui exprime que la machine à environnement peut passer du premier état au second par un nombre arbitraire d'étapes de réduction.

```
Inductive multireduce : expr * cont -> expr * cont -> Prop :=
| start : forall ek, multireduce ek ek
| step1 : forall e1 e2 k ek, multireduce (e1, AddL e2 k) ek -> multireduce (Add e1 e2, k) ek
| step2 : forall e1 e2 k ek, multireduce (e1, MulL e2 k) ek -> multireduce (Mul e1 e2, k) ek
| step3 : forall n e k ek, multireduce (e, AddR n k) ek -> multireduce (Cst n, AddL e k) ek
| step4 : forall n m k ek, multireduce (Cst(m+n), k) ek -> multireduce (Cst n, AddR m k) ek
| step5 : forall n e k ek, multireduce (e, MulR n k) ek -> multireduce (Cst n, MulL e k) ek
| step6 : forall n m k ek, multireduce (Cst(m*n), k) ek -> multireduce (Cst n, MulR m k) ek.
```

D – Certification On prouve que la machine évalue correctement.

6. Formuler un énoncé Coq spécifiant que la machine abstraite évalue correctement les expressions, c'est-à-dire que pour toute expression e il existe un nombre d'opérations au bout desquelles `compute` renvoie un résultat qui est exactement la valeur de l'expression telle que calculée par `value`.

   ```
   Theorem spec : forall e, exists n, compute e n = Some (value e).
   ```

7. Formuler une autre spécification de la correction de la machine reposant cette fois sur `multireduce`.

   ```
   Theorem spec' : forall e, exists v, multireduce (e, End) (Cst v, End).
   ```

8. Donner *informellement* les grandes lignes de la preuve de correction, en utilisant au choix soit la spécification 6. soit la spécification 7.

   Dans les deux cas, on peut calculer une mesure `phi (e,k)$ des états qui est la somme d'une mesur` et d'une mesure de `$` Pour `e`, `Cst` compte 0 tandis que `Add` and `Mul` compte 3. Pour `k`, `End` compte 0 tandis que `AddL` and `MulL` compte 2 et `AddR` et `MulR` compte 1.

   Pour la première spécification, on définit aussi une fonction `value_cont : nat -> cont -> nat` qui évalue l'effet d'une continuation sur une valeur (par exemple `value_cont n (AddL e k)` est `value_cont (n+value e) k`). On montre alors le lemme plus général

   ```
   Lemma gen_spec : forall e k, tail (exec' (e, k)) (phi (e, k)) = Some (value_cont (value
   ```

   par analyse de cas sur l'expression et la continuation.

   Dans le second cas, on montre le lemme plus général

   ```
   Lemma gen_spec' : forall ek, exists v, multireduce ek (Cst v, End)
   ```

par induction sur `phi ek` et analyse de cas sur l'expression et sur la continuation avant de l'instancier à `(e, End)`.

Rappels : l'addition et la multiplication sur `nat` se nomment `Nat.add` (notation « `+` ») et `Nat.mul` (notation « `*` »).