INF110 / CSC-3TC34-TP
Contrôle de connaissances — Corrigé

Logique et Fondements de l'Informatique

29 janvier 2025

Consignes.

Les exercices et le problème sont totalement indépendants les uns des autres. Ils pourront être traités dans un ordre quelconque, mais on demande de faire apparaître de façon très visible dans les copies où commence chaque exercice (tirez au moins un trait sur toute la largeur de la feuille entre deux exercices).

Les questions du problème dépendent les unes des autres, mais ont été rédigées de manière à ce que chacune donne toutes les informations nécessaires pour passer à la suite. Mais comme elles (les questions du problème) présentent une gradation approximative de difficulté, il est recommandé de les traiter dans l'ordre.

La longueur du sujet ne doit pas effrayer : l'énoncé du problème est très long parce que des rappels et éclaircissements ont été faits et que les questions ont été rédigées de façon aussi précise que possible. Par ailleurs, il ne sera pas nécessaire de tout traiter pour avoir le maximum des points.

Dans les exercices portant sur Coq (exercices 1 à 4), les erreurs de syntaxe Coq ne seront pas pénalisées tant qu'on comprend clairement l'intention.

L'usage de tous les documents écrits (notes de cours manuscrites ou imprimées, feuilles d'exercices, livres) est autorisé.

L'usage des appareils électroniques est interdit.

Durée : 3h

Ce corrigé comporte 15 pages (page de garde incluse).

Git : 18f8b91 Thu Jan 23 22:39:25 2025 +0100

Exercice 1.

Dans cet exercice, on considère des paires d'états d'une preuve en Coq avant et après l'application d'une tactique. On demande de retrouver quelle est la tactique ou la séquence de tactiques appliquée.

(1) On part de l'état suivant :

```
  A, B, C : Prop
  ============================
  (A /\ B) /\ C <-> A /\ B /\ C
```

et on veut arriver à l'état suivant :

```
First subgoal:

  A, B, C : Prop
  ============================
  (A /\ B) /\ C -> A /\ B /\ C

Second subgoal:

  A, B, C : Prop
  ============================
  A /\ B /\ C -> (A /\ B) /\ C
```

(2) On part de l'état suivant :

```
  A, B, C : Prop
  H : (A /\ B) /\ C
  ============================
  A
```

et on veut arriver à l'état suivant :

```
  H1 : A
  H2 : B
  H3 : C
  ============================
  A
```

(3) On part de l'état suivant :

```
  A, B, C : Prop
  H : A
  ============================
  A \/ B
```

et on veut arriver à l'état suivant :

```
  No more goals.
```

(4) On part de l'état suivant :

```
  ============================
  (A -> B) -> ~ B -> ~ A
```

et on veut arriver à l'état suivant :

```
  H1 : A -> B
  H2 : ~ B
  H3 : A
  ============================
  False
```

(5) On part de l'état suivant :

```
  n, m : nat
  ============================
  n + m = m + n
```

et on veut arriver à l'état suivant :

```
First subgoal:

  m : nat
  ============================
  0 + m = m + 0

Second subgoal:

  n, m : nat
  IHn : n + m = m + n
  ============================
  S n + m = m + S n
```

(6) On part de l'état suivant :

```
  n, m : nat
  ============================
  S (n + m) = S (m + n)
```

et on veut arriver à l'état suivant :

```
  n, m : nat
  ============================
  n + m = m + n
```

(7) On part de l'état suivant :

```
n, m : nat
H : S n = S m
============================
n + 1 = m + 1
```

et on veut arriver à l'état suivant :

```
n, m : nat
H : n = m
============================
n + 1 = m + 1
```

Corrigé.

(1) `split.`

(2) En deux étapes : `destruct H as (H1, H3). destruct H1 as (H1, H2).` Possible également en une seule étape : `destruct H as ((H1, H2), H3).`

(3) `left. assumption.`

(4) `intros H1 H2 H3.`

(5) `induction n as [|n IHn].` ou simplement `induction n.`

(6) `f_equal.` (la tactique) ou `apply f_equal.` (le lemme).

(7) `injection H as H.` ✓

Exercice 2.

Si l'on dispose du lemme suivant en Coq :

```
Lemma add_0_r : forall n : nat, n + 0 = n.
```

Parmi les buts suivants, quand peut-on utiliser ce lemme avec la tactique `rewrite` ? Quand peut-on utiliser ce lemme avec la tactique `apply` ? Justifier brièvement.

(1)

```
n : nat
============================
n + 0 = 0 + n
```

(2)

```
n, m : nat
============================
(n + m) + 0 = m + n
```

(3)

```
n, m : nat
============================
(m * n) + 0 = m * n
```

(4)

```
n, m : nat
============================
(n + 0) + m = n + m
```

Corrigé.

(1) On peut utiliser `rewrite -> add_0_r.` car le but contient un sous-terme de la forme `?n + 0` (où `?n` est `n`). On peut également utiliser `apply add_0_r.` car le but est convertible à `?n + 0 = ?n` (car la partie droite de l'égalité se simplifie en `n`).

(2) On peut utiliser `rewrite -> add_0_r.` car le but contient un sous-terme de la forme `?n + 0` (où `?n` est `n + m`). On ne peut pas utiliser `apply add_0_r.` car le but n'est pas convertible à `?n + 0 = ?n` (sauf à réécrire au préalable le but avec la commutativité de l'addition).

(3) On peut utiliser `rewrite -> add_0_r.` car le but contient un sous-terme de la forme `?n + 0` (où `?n` est `m * n`). On peut également utiliser `apply add_0_r.` car le but est de la forme `?n + 0 = ?n` (où `?n` est `m * n`).

(4) On peut utiliser `rewrite -> add_0_r.` car le but contient un sous-terme de la forme `?n + 0` (où `?n` est `n`). On ne peut pas utiliser `apply add_0_r.` car le but n'est pas convertible à `?n + 0 = ?n` (sauf à se débarrasser au préalable du `+ m` grâce à la tactique `f_equal`). ✓

Exercice 3.

(A) Dans cet exercice, on donne un terme de preuve Coq et on demande de retrouver le théorème du calcul propositionnel intuitionniste qu'il prouve.

(1) `fun (H : A) => H`

(2) `fun (H1 : A -> B) (H2 : B -> C) (H3 : A) => H2 (H1 H3)`

(3) `fun (H1 : A -> B) (H2 : ~ B) => (fun (H3 : A) => H2 (H1 H3))`

(B) Écrire la démonstration décrite par le terme (3) en déduction naturelle (on donnera l'arbre de preuve ou la présentation en style drapeau, comme on préfère).

Corrigé.

(A) (1) $A \Rightarrow A$.

(2) $(A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow A \Rightarrow C$.

(3) $(A \Rightarrow B) \Rightarrow \neg B \Rightarrow \neg A$.

(B) Voici la démonstration de (3) écrite en arbre de preuve :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(A \Rightarrow B), \neg B, A \vdash \neg B}\,\text{Ax} \quad \dfrac{\dfrac{}{(A \Rightarrow B), \neg B, A \vdash A \Rightarrow B}\,\text{Ax} \quad \dfrac{}{(A \Rightarrow B), \neg B, A \vdash A}\,\text{Ax}}{(A \Rightarrow B), \neg B, A \vdash B}\,\Rightarrow\text{Élim}}{(A \Rightarrow B), \neg B, A \vdash \bot}\,\Rightarrow\text{Élim}}{(A \Rightarrow B), \neg B \vdash A \Rightarrow \bot}\,\Rightarrow\text{Int}}{A \Rightarrow B \vdash \neg B \Rightarrow \neg A}\,\Rightarrow\text{Int}}{\vdash (A \Rightarrow B) \Rightarrow \neg B \Rightarrow \neg A}\,\Rightarrow\text{Int}$$

ou avec uniquement les conclusions de chaque séquent, et en indiquant les noms comme dans le

terme Coq :

$$\dfrac{\dfrac{H_2\ \overline{\neg B} \qquad \dfrac{H_1\ \overline{A \Rightarrow B} \qquad \overline{A}\ H_3}{B}\ \Rightarrow\text{ÉLIM}}{\bot}\ \Rightarrow\text{ÉLIM}}{\dfrac{A \Rightarrow \bot}{\neg B \Rightarrow \neg A}\ \Rightarrow\text{INT}(H_2)}\ \Rightarrow\text{INT}(H_3)}{(A \Rightarrow B) \Rightarrow \neg B \Rightarrow \neg A}\ \Rightarrow\text{INT}(H_1)$$

et la voici écrite dans le style « drapeau » :

(1)	$\boxed{A \Rightarrow B}$	
(2)	$\quad\boxed{\neg B}$	
(3)	$\quad\quad\boxed{A}$	
(4)	$\quad\quad B$	$\Rightarrow$Élim sur (1) et (3)
(5)	$\quad\quad \bot$	$\Rightarrow$Élim sur (2) et (4)
(6)	$\quad A \Rightarrow \bot$	$\Rightarrow$Int de (3) dans (5)
(7)	$\neg B \Rightarrow \neg A$	$\Rightarrow$Int de (2) dans (6)
(8)	$(A \Rightarrow B) \Rightarrow \neg B \Rightarrow \neg A$	$\Rightarrow$Int de (1) dans (8)

✓

Exercice 4.

(1) Définir en Coq un type pour représenter les jours de la semaine (7 valeurs possibles).

(2) Définir une fonction qui, étant donné un jour de la semaine, renvoie le jour suivant.

(3) Énoncer un lemme en Coq affirmant que le jour suivant du dimanche est le lundi.

(4) Avec quelle(s) tactique(s) peut-on prouver ce lemme ?

(5) Énoncer un lemme en Coq affirmant que le jour suivant du jour j n'est pas j.

(6) Expliquer dans les grandes lignes comment on pourrait prouver ce lemme en Coq (notamment quelles seraient les principales tactiques utilisées).

Corrigé.

(1) On définit un type inductif à 7 constructeurs :

```
Inductive jour : Type :=
| lundi : jour
| mardi : jour
| mercredi : jour
| jeudi : jour
| vendredi : jour
| samedi : jour
| dimanche : jour.
```

(2) On définit une fonction (non récursive) par pattern matching :

```
Definition suivant (j : jour) : jour :=
  match j with
  | lundi => mardi
  | mardi => mercredi
  | mercredi => jeudi
  | jeudi => vendredi
  | vendredi => samedi
  | samedi => dimanche
  | dimanche => lundi
  end.
```

(3) `Lemma suivant_dimanche_lundi : suivant dimanche = lundi.`

(4) `simpl.` (optionnel) suivi de `reflexivity.`

(5) `Lemma suivant_est_different : forall j : jour, suivant j <> j.` ou `forall j : jour, ~ suivant j = j` (équivalents étant donné que `<>` n'est qu'une notation en Coq).

(6) On peut prouver ce lemme par analyse de cas sur `j` (tactique `destruct j.`). Après un appel optionnel à `simpl.` ou `simpl in *.` pour calculer la valeur de `suivant j` dans chaque cas, on peut conclure par `discriminate.` pour montrer que les deux termes sont différents.

Preuve complète :

```
Lemma suivant_est_different : forall j : jour, suivant j <> j.
Proof.
  intros j H.
  destruct j; simpl in *; discriminate.
Qed.
```

✓

Exercice 5.

En utilisant une des sémantiques vues en cours pour le calcul propositionnel intuitionniste, montrer que la formule suivante n'est pas démontrable en calcul propositionnel intuitionniste :

$$\big(((A \Rightarrow B) \Rightarrow B) \wedge ((B \Rightarrow A) \Rightarrow A)\big) \;\Rightarrow\; (A \vee B)$$

(*Indications :* Si on souhaite utiliser la sémantique des ouverts, on pourra utiliser deux demi-droites de même origine dans la droite réelle. Si on souhaite utiliser la sémantique de Kripke, on pourra prendre le cadre $\{u, v, w\}$ avec l'ordre engendré par les seules relations $u \leq v$ et $u \leq w$, et considérer les affectations de vérité valant 1 uniquement dans le monde v respectivement w. Si on souhaite utiliser la sémantique de la réalisabilité propositionnelle, on remarquera qu'un programme e qui réalise cette formule pour $A = \mathbb{N}$ et $B = \varnothing$ doit renvoyer un élément de la forme $\langle 0, — \rangle$ si on l'applique à $\langle c, c \rangle$ avec c le code de Gödel d'une fonction constante quelconque, et que pour $A = \varnothing$ et $B = \mathbb{N}$ il doit renvoyer $\langle 1, — \rangle$.)

La formule en question est-elle démontrable en calcul propositionnel classique ? (On ne demande pas forcément d'en exhiber une démonstration.)

Corrigé. *Avec la sémantique des ouverts :* Travaillons sur les ouverts de $X = \mathbb{R}$. Posons $U = \{t \in \mathbb{R} : t < 0\}$ et $V = \{t \in \mathbb{R} : t > 0\}$. Ce sont bien des ouverts. Alors $(U \mathrel{\dot{\Rightarrow}} V) = V$ donc $((U \mathrel{\dot{\Rightarrow}} V) \mathrel{\dot{\Rightarrow}} V) = \mathbb{R}$. Symétriquement, $((V \mathrel{\dot{\Rightarrow}} U) \mathrel{\dot{\Rightarrow}} U) = \mathbb{R}$. En revanche, $U \mathrel{\dot{\vee}} V = \mathbb{R} \setminus \{0\}$. On

en déduit pour la formule tout entière que $\big(((U \mathrel{\dot\Rightarrow} V) \mathrel{\dot\Rightarrow} V) \wedge ((V \mathrel{\dot\Rightarrow} U) \mathrel{\dot\Rightarrow} U)\big) \mathrel{\dot\Rightarrow} (U \vee B)$ vaut $\mathbb{R} \setminus \{0\}$: comme ceci n'est pas X tout entier, la formule n'est pas prouvable (par correction de la sémantique des ouverts de X).

Avec la sémantique de Kripke : Travaillons dans le cadre de Kripke $\{u, v, w\}$ avec pour seules relations non-triviales $u \leq v$ et $u \leq w$. Soit p l'affectation de vérité valant 1 en v et 0 partout ailleurs, et q celle valant 1 en w et 0 partout ailleurs. Ce sont bien des affectations de vérité (c'est-à-dire ici que si elles valent 1 en u, ce qui ne se produit jamais, elles valent 1 partout). On vérifie alors que $(p \dot\Rightarrow q) = q$ donc $((p \dot\Rightarrow q) \dot\Rightarrow q) = 1$. Symétriquement, $((q \dot\Rightarrow p) \dot\Rightarrow p) = 1$. En revanche, $p \dot\vee q$ vaut 1 en v et w mais 0 en u. On en déduit pour la formule tout entière que $\big(((p \dot\Rightarrow q) \dot\Rightarrow q) \wedge ((q \dot\Rightarrow p) \dot\Rightarrow p)\big) \mathrel{\dot\Rightarrow} (p \vee q)$ vaut 1 en v et w mais 0 en u : comme ceci n'est pas 1 partout, la formule n'est pas prouvable (par correction de la sémantique de Kripke).

Avec la sémantique de la réalisabilité propositionnelle : Supposons que e réalise la formule. Si $A = \mathbb{N}$ et $B = \varnothing$, alors $(A \mathrel{\dot\Rightarrow} B) = \varnothing$ donc $((A \mathrel{\dot\Rightarrow} B) \mathrel{\dot\Rightarrow} B) = \mathbb{N}$, et $(B \mathrel{\dot\Rightarrow} A) = \mathbb{N}$ donc $((A \dot\Rightarrow B) \dot\Rightarrow B) = \mathbb{N} \dot\Rightarrow \mathbb{N}$ est l'ensemble des codes de Gödel des fonctions récursives totales $\mathbb{N} \to \mathbb{N}$. Fixons c un tel code de Gödel, disons celui du programme qui renvoie immédiatement 0. Alors $\langle c, c\rangle$ est dans $((A \mathrel{\dot\Rightarrow} B) \mathrel{\dot\Rightarrow} B) \mathrel{\dot\wedge} ((A \mathrel{\dot\Rightarrow} B) \mathrel{\dot\Rightarrow} B)$, donc $\varphi_e(\langle c, c\rangle)$ doit être dans $A \mathrel{\dot\vee} B$, et comme B est vide, il est forcément de la forme $\langle 0, m\rangle$ avec $m \in A$. Mais exactement le même raisonnement en échangeant A et B (qui ont un rôle symétrique) montre que si $A = \varnothing$ et $B = \mathbb{N}$ alors $\varphi_e(\langle c, c\rangle)$ doit être dans $A \mathrel{\dot\vee} B$ et cette fois de la forme $\langle 1, n\rangle$ avec $n \in B$. Comme $\varphi_e(\langle c, c\rangle)$ ne peut pas être à la fois de la forme $\langle 0, —\rangle$ et $\langle 1, —\rangle$, c'est que la formule n'est pas réalisable. En particulier, elle n'est pas prouvable (par correction de la sémantique de la rélisabilité propositionnelle).

Enfin, concernant la formule en logique classique, un simple tableau de vérité montre qu'elle est vraie dans chacun des quatre cas de figure, donc la formule est classiquement prouvable (par complétude de la sémantique booléenne pour la logique classique). ✓

Problème 6.

Notations : Dans tout cet exercice, on notera comme d'habitude $\varphi_e \colon \mathbb{N} \dashrightarrow \mathbb{N}$ (pour $e \in \mathbb{N}$) la e-ième fonction partielle calculable (i.e. générale récursive). On notera

$$\mathsf{PartCalc} := \{\varphi_e \ : \ e \in \mathbb{N}\} = \{g \colon \mathbb{N} \dashrightarrow \mathbb{N} \ : \ \exists e \in \mathbb{N}.\, (g = \varphi_e)\}$$

l'ensemble des fonctions *partielles* calculables $\mathbb{N} \dashrightarrow \mathbb{N}$, ainsi que

$$\mathsf{TotCode} := \{e \in \mathbb{N} \ : \ \varphi_e \text{ est totale}\} = \{e \in \mathbb{N} \ : \ \varphi_e \in \mathsf{TotCalc}\}$$

l'ensemble des codes des programmes qui définissent une fonction *totale* $\mathbb{N} \to \mathbb{N}$ (i.e., terminent et renvoient un entier quel que soit l'entier qu'on leur fournit en entrée) [1], et

$$\mathsf{TotCalc} := \{\varphi_e \ : \ e \in \mathsf{TotCode}\} = \{g \colon \mathbb{N} \to \mathbb{N} \ : \ \exists e \in \mathbb{N}.\, (g = \varphi_e)\}$$

l'ensemble correspondant des fonctions *totales* calculables $\mathbb{N} \to \mathbb{N}$, i.e., celles calculées par les programmes qu'on vient de dire.

On va s'intéresser à la notion (qu'on va définir) de fonction calculable $\mathsf{PartCalc} \to \mathbb{N}$ d'une part, et $\mathsf{TotCalc} \to \mathbb{N}$ d'autre part. (On parle parfois de « fonctionnelles » ou de « fonction de second ordre » pour ces fonctions sur les fonctions. On souligne qu'on parle bien ici de fonction *totales* $\mathsf{PartCalc} \to \mathbb{N}$ et $\mathsf{TotCalc} \to \mathbb{N}$.)

Définition : (A) Une fonction (totale !) $F \colon \mathsf{PartCalc} \to \mathbb{N}$ est dite *calculable* lorsque la fonction $\hat{F} \colon \mathbb{N} \to \mathbb{N}$ définie par $\hat{F}(e) = F(\varphi_e)$ est calculable. (B) De même, une fonction (totale) $F \colon \mathsf{TotCalc} \to \mathbb{N}$ est dite calculable lorsqu'il existe une fonction $\hat{F} \colon \mathbb{N} \dashrightarrow \mathbb{N}$ partielle calculable telle que telle que $\hat{F}(e) = F(\varphi_e)$ pour tout $e \in \mathsf{TotCode}$.

En français : une fonctionnelle définie sur l'ensemble des fonctions calculables partielles ou totales est elle-même dite calculable lorsqu'il existe un programme qui calcule $F(g)$ à partir du code e d'un programme quelconque calculant g.

Dans le cas (A), la définition implique que la fonction $\hat{F}$ vérifie forcément $(\varphi_{e_1} = \varphi_{e_2}) \implies (\hat{F}(e_1) = \hat{F}(e_2))$; c'est-à-dire qu'elle est « extensionnelle » : elle doit renvoyer la même valeur sur deux programmes qui calculent la même fonction (= ont la même « extension »). D'ailleurs (on ne demande pas de justifier ce fait), se donner une fonction $F \colon \mathsf{PartCalc} \to \mathbb{N}$ revient exactement à se donner une fonction $\hat{F} \colon \mathbb{N} \to \mathbb{N}$ ayant cette propriété d'« extensionnalité ». (De même, dans le cas (B), se donner une fonction $F \colon \mathsf{TotCalc} \to \mathbb{N}$ revient exactement à se donner une fonction qui soit extensionnelle sur $\mathsf{TotCode}$.) La définition ci-dessus dit donc que F est calculable lorsque la $\hat{F}$ qui lui correspond est elle-même calculable dans le cas (A), ou, dans le cas (B) la restriction à $\mathsf{TotCode}$ d'une fonction calculable partielle sur $\mathbb{N}$ dont le domaine de définition contient au moins $\mathsf{TotCode}$.

(1) Montrer que toute fonction (totale !) calculable $F \colon \mathsf{PartCalc} \to \mathbb{N}$ est en fait constante.

Pour cela, on pourra supposer par l'absurde que F prend deux valeurs distinctes, disons v_1 et v_2, et considérer l'ensemble des e tels que $F(\varphi_e) = v_1$ (i.e., tels que $\hat{F}(e) = v_1$). (Pourquoi est-il décidable ? Et pourquoi est-ce une contradiction ?)

Corrigé. Si F n'est pas constante, il existe $v_1 \neq v_2$ tels que F prenne la valeur v_1 et la valeur v_2, disons $F(\varphi_{e_1}) = v_1$ et $F(\varphi_{e_2}) = v_2$. Considérons l'ensemble D_1 des e tels que $F(\varphi_e) = v_1$, c'est-à-dire $\hat{F}(e) = v_1$ par définition de $\hat{F}$. D'une part, cet ensemble D_1 est décidable : pour décider si $e \in D_1$, il suffit de calculer $\hat{F}(e)$ (ce qui est possible par l'hypothèse que F, i.e., que $\hat{F}$, est

1. Par souci de cohérence, on pourrait aussi vouloir définir $\mathsf{PartCode} = \mathbb{N}$ comme l'ensemble des codes des programmes définissant une fonction partielle : nous suivons la convention que toute erreur dans un programme (même « syntaxique ») conduit à une valeur non-définie.

calculable), et tester si sa valeur vaut v_1. D'autre part, D_1 n'est ni vide (puisque $e_1 \in D_1$) et n'est pas plein (puisque $e_2 \notin D_1$). Or ceci contredit le théorème de Rice, lequel affirme que si $E \subseteq \mathsf{PartCalc}$ est tel que l'ensemble $\{e \in \mathbb{N} : \varphi_e \in E\}$ est décidable, alors il est vide ou plein (ici, on prend $E = \{g \in \mathsf{PartCalc} : F(g) = v_1\}$). ✓

(2) Expliquer pourquoi il existe des fonctions calculables (totales) $F\colon \mathsf{TotCalc} \to \mathbb{N}$ qui ne sont pas constantes. Donner un exemple explicite.

Pour cela, on pourra penser évaluer en un point, c'est-à-dire exécuter, le programme dont on a reçu le code en argument (on rappellera pourquoi on peut faire cela). Par ailleurs, on fera clairement ressortir pourquoi le raisonnement tenu ici ne s'applique pas à la question (1).

Corrigé. La fonction partielle $e \mapsto \varphi_e(0)$ (ou, plus généralement, $(e, x) \mapsto \varphi_e(x)$) est calculable par l'existence d'un programme universel. Ceci montre que la fonction $F\colon \mathsf{TotCalc} \to \mathbb{N}$ donnée par $g \mapsto g(0)$ est calculable (la fonction $\hat{F}$ qui lui correspond étant justement $e \mapsto \varphi_e(0)$). Ce raisonnement n'était pas applicable à PartCalc car on cherche à définir une fonction *totale*, or $g \mapsto g(0)$ n'est que partielle sur PartCalc (tandis qu'elle est totale sur TotCalc).

Plus généralement, on peut construire énormément de fonctions calculables $\mathsf{TotCalc} \to \mathbb{N}$ en appliquant librement l'évaluation de l'argument qu'on a reçu (qui, par hypothèse, est toujours défini). Un autre exemple serait la fonction $g \mapsto g(0) + g(1)$ ou encore $g \mapsto g(g(0))$. ✓

Fixons maintenant une fonction calculable $F\colon \mathsf{TotCalc} \to \mathbb{N}$, ainsi que la fonction $\hat{F}\colon \mathbb{N} \dashrightarrow \mathbb{N}$ qui lui correspond d'après le (B) des définitions ci-dessus.

Dans les questions (3) à (8) qui suivent, on cherche à démontrer que F a la propriété[2] suivante (« théorème de Kreisel-Lacombe-Shoenfield ») : quelle que soit $g \in \mathsf{TotCalc}$, il existe n tel que pour toute fonction $g' \in \mathsf{TotCalc}$ qui coïncide avec g jusqu'au rang n (i.e. : $\forall i \leq n.\, (g'(i) = g(i))$) on ait $F(g') = F(g)$.

Notations : Soit NulAPCR l'ensemble des fonctions $h\colon \mathbb{N} \to \mathbb{N}$ qui sont nulles à partir d'un certain rang, c'est-à-dire telles que $\exists m. \forall i \geq m.\, (h(i) = 0)$.

(3) (a) Expliquer pourquoi $\mathsf{NulAPCR} \subseteq \mathsf{TotCalc}$, i.e., pourquoi toute fonction $\mathbb{N} \to \mathbb{N}$ nulle à partir d'un certain rang est automatiquement calculable. **(b)** Montrer, plus précisément, qu'il existe une fonction calculable $\Gamma\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ telle que les fonctions $h \in \mathsf{NulAPCR}$ soient exactement celles de la forme $\Gamma(j, —)$ (c'est-à-dire $i \mapsto \Gamma(j, i)$) pour un certain j. (*Indication :* on pourra utiliser un codage de Gödel des suites finies d'entiers naturels par des entiers naturels j ; on ne demande pas de justifier les détails de ce codage.) **(c)** En déduire qu'il existe $\gamma\colon \mathbb{N} \to \mathbb{N}$ calculable telle que les fonctions de NulAPCR soient exactement celles de la forme $\varphi_{\gamma(j)}$ pour un certain j. (*Indication :* on pourra utiliser le théorème s-m-n.)

Corrigé. **(a)** Si on a $h(i) = 0$ pour $i \geq m$, alors h est trivialement calculable par le programme « si $i = 0$, renvoyer $h(0)$, si $i = 1$ renvoyer $h(1)$, ..., si $i = m - 1$ renvoyer $h(m-1)$, et si $i \geq m$ renvoyer 0 ».

(b) On considère le programme qui, donné deux entiers j, i effectue les opérations suivantes : il décode j comme le codage de Gödel $\langle\!\langle a_0, \ldots, a_{m-1} \rangle\!\rangle$ d'un m-uplet $(a_0, \ldots, a_{m-1})$ d'entiers naturels, puis, si $i < m$, renvoie a_i, et sinon, renvoie 0. Ce programme calcule une fonction $\Gamma\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. Et par construction, chaque fonction $\Gamma(j, —)$ est nulle à partir d'un certain rang, et chaque fonction h qui est nulle à partir du rang m est de la forme $\Gamma(j, —)$, à savoir pour $j = \langle\!\langle h(0), \ldots, h(m-1) \rangle\!\rangle$.

2. Si on sait ce que cela signifie, cette propriété peut s'exprimer ainsi : F est continue (ou, ce qui revient ici au même : localement constante) lorsque TotCalc est muni de la topologie [héritée de la topologie] produit sur $\mathbb{N}^{\mathbb{N}}$.

(c) Le théorème s-m-n assure qu'on peut calculer le code $\gamma(j)$ d'un programme calculant $\Gamma(j,—)$ de manière calculable à partir de j. (De façon extrêmement précise : si p est un code tel que $\varphi_p^{(2)}(j,i) = \Gamma(j,i)$, alors $\varphi^{(1)}_{s_{1,1}(p,j)}(i) = \varphi_p^{(2)}(j,i) = \Gamma(j,i)$, donc $\gamma(j) := s_{1,1}(p,j)$ convient au sens où $\varphi_{\gamma(j)} = \Gamma(j,—)$. La fonction γ est bien calculable car $s_{1,1}$ l'est ; elle est même primitive récursive.) ✓

Notations : Si $g \in \mathsf{TotCalc}$ et $n \in \mathbb{N}$, on appellera $\mathcal{B}(g,n) := \{g' \in \mathsf{TotCalc} : \forall i \leq n.\, (g'(i) = g(i))\}$ l'ensemble des g' qui coïncident avec g jusqu'au rang n, et $\mathcal{B}_0(g,n) := \mathcal{B}(g,n) \cap \mathsf{NulAPCR}$ celles qui, en outre, sont nulles à partir d'un certain rang (c'est-à-dire les fonctions prenant des valeurs de la forme $g(0), g(1), \ldots, g(n), ?, ?, \ldots, ?, 0, 0, 0, \ldots$).

(On rappelle que notre but est de montrer qu'il existe n tel que F soit constante sur $\mathcal{B}(g,n)$.)

(4) Soit $g \in \mathsf{TotCalc}$ et $n \in \mathbb{N}$. Expliquer *brièvement* pourquoi les conclusions de la question (3) valent encore en remplaçant $\mathsf{NulAPCR}$ par $\mathcal{B}_0(g,n)$ partout. (*Indication :* on peut par exemple fabriquer un élément de $\mathcal{B}_0(g,n)$ en insérant les valeurs $g(0), \ldots, g(n)$ avant un élément de $\mathsf{NulAPCR}$.) En particulier : on notera $\gamma(g,n,j)$ la conclusion de la dernière sous-question, c'est-à-dire que les fonctions $h \in \mathcal{B}_0(g,n)$ soient exactement celles de la forme $h = \varphi_{\gamma(g,n,j)}$ pour un certain j ; on expliquera *brièvement* pourquoi $\gamma(g,n,j)$ est calculable en fonction de j, de n et du code (dans $\mathsf{TotCode}$) d'un programme calculant g.

Corrigé. En (3)(b), on a vu qu'on pouvait fabriquer n'importe quel élément de $\mathsf{NulAPCR}$ en prenant le codage j d'un m-uplet qu'on étend en la fonction qui commence par ces m valeurs puis une infinité de zéros. Pour obtenir tous les éléments de $\mathcal{B}_0(g,n)$, il suffit par exemple d'insérer $g(0), \ldots, g(n)$ avant les valeurs codées par j. Il faut simplement justifier que le calcul des valeurs $g(0), \ldots, g(n)$ (ou du tuple de ces valeurs) est faisable en fonction de n et du code d'un programme calculant g, or c'est clair (par l'existence d'un programme universel). ✓

Si on n'a pas su traiter les questions (3) et/ou (4), pour la suite, **on retiendra juste ceci :** $\gamma(g,n,j)$ est calculable, et on a $\mathcal{B}_0(g,n) = \{\varphi_{\gamma(g,n,j)} : j \in \mathbb{N}\}$.

Algorithme A : Pour $g \in \mathsf{TotCalc}$, on considère maintenant le programme prenant deux entrées d et ℓ, décrit informellement ainsi :

— lancer l'exécution de $\varphi_d(0)$ pour au plus ℓ étapes [3] ;
— si l'exécution ne s'est pas terminée dans le temps imparti, terminer en renvoyant la valeur $g(\ell)$;
— sinon, disons si l'exécution s'est terminée en exactement $n \leq \ell$ étapes, lancer une boucle non bornée pour rechercher un $j \in \mathbb{N}$ tel que [4] $\hat{F}(\gamma(g,n,j)) \neq F(g)$:
  — si un tel j est trouvé, on renvoie $\varphi_{\gamma(g,n,j)}(\ell)$,
  — sinon, bien sûr, l'algorithme ne termine pas.

On notera $g_d(\ell)$ la valeur calculée par cet algorithme A (s'il termine).

(5) (a) Justifier que les opérations présentées dans l'algorithme A ont bien un sens (i.e., que c'est bien un algorithme, qu'on a bien défini une fonction $\mathbb{N} \times \mathbb{N} \dashrightarrow \mathbb{N}$ partielle calculable $(d,\ell) \mapsto g_d(\ell)$). **(b)** En déduire qu'il existe une fonction calculable totale $e \mapsto e_d$ telle que $g_d = \varphi_{e_d}$ pour tout d.

Corrigé. **(a)** Parmi les points pouvant mériter d'être justifiés :

3. Si on préfère la notion d'arbre de calcul, remplacer par : « rechercher parmi les entiers $n \leq \ell$ le code d'un arbre de calcul de $\varphi_d(0)$ », et de même plus loin, remplacer « l'exécution s'est terminée en exactement $n \leq \ell$ étapes » par « n est le code d'un arbre de calcul de $\varphi_d(0)$ ». Cela ne changera rien au problème.

4. On rappelle à toutes fins utiles que $\hat{F}(\gamma(g,n,j)) = F(\varphi_{\gamma(g,n,j)})$ (c'est la définition de $\hat{F}$).

— il est bien possible de lancer l'exécution de $\varphi_d(0)$ pour au plus ℓ étapes (programme universel avec limite de temps / théorème de la forme normale de Kleene) ;
— renvoyer $g(\ell)$ ne pose pas de problème car g est calculable ;
— calculer $\hat{F}(\gamma(g,n,j))$ est possible car γ est calculable (question (4)) et $\hat{F}$ l'est (par définition de la calculabilité de F) ;
— calculer $F(g)$ est possible, là aussi car F est calculable (note : à la limite, ce point se passe de justification car $F(g)$ est de toute façon une constante dans notre programme, sauf si on fait varier g, ce qui ne sera le cas qu'à la question (9)) ;
— on a bien le droit de faire une boucle non bornée sur j par l'opérateur μ de Kleene ;
— calculer $\varphi_{\gamma(g,n,j)}(\ell)$ est possible car γ est calculable (et en utilisant un programme universel pour le φ) ; d'ailleurs, ce calcul termine forcément car $\varphi_{\gamma(g,n,j)}$ est une fonction totale par définition de $\gamma(g,n,j)$.

(b) On utilise le théorème s-m-n exactement comme en (3)(c) : dès lors que $(d,\ell) \mapsto g_d(\ell)$ est calculable, on en déduit une fonction $d \mapsto e_d$ calculable (totale ! et même primitive récursive, à savoir $e_d = s_{1,1}(q,d)$ où q est tel que $g_d(\ell) = \varphi_q^{(2)}(d,\ell)$) vérifiant $g_d = \varphi_{e_d}$. ✓

(6) Soit $\mathscr{H} := \{d \in \mathbb{N} : \varphi_d(0)\!\downarrow\}$ l'ensemble des d tels que l'exécution de $\varphi_d(0)$ termine. **(a)** Lorsque $d \notin \mathscr{H}$, que vaut g_d ? et du coup, que vaut $\hat{F}(e_d)$? **(b)** Montrer que $\{d \in \mathbb{N} : \hat{F}(e_d) = F(g)\}$ est semi-décidable. **(c)** Le complémentaire $\complement\mathscr{H} = \{d \in \mathbb{N} : \varphi_d(0) \uparrow\}$ de $\mathscr{H}$ est-il semi-décidable ? **(d)** En déduire qu'il existe $d \in \mathscr{H}$ tel que $\hat{F}(e_d) = F(g)$.

Corrigé. **(a)** Si $d \notin \mathscr{H}$, alors dans l'algorithme A, on est toujours dans le cas « l'exécution ne s'est pas terminée dans le temps imparti », donc $g_d(\ell)$ pour tout ℓ, c'est-à-dire que $g_d = g$ (et notamment, g_d est *totale*). Par conséquent, $\hat{F}(e_d) = F(g_d) = F(g)$ (par extensionnalité).

(b) Pour semi-décider si $\hat{F}(e_d) = F(g)$, il suffit de lancer le calcul de $\hat{F}(e_d)$ et, si celle-ci termine, comparer son résultat à $F(g)$ et renvoyer vrai si c'est le cas (et sinon, faire une boucle infinie).

(On notera que le calcul de $\hat{F}(e_d)$ ne termine pas forcément, parce qu'on n'a pas la certitude que $e_d \in \mathsf{TotCode}$ vu qu'on ne sait pas si g_d est totale.)

(c) L'ensemble $\mathscr{H}$ n'est pas décidable (c'est une variante du problème de l'arrêt, ou bien on peut invoquer le théorème de Rice) ; or il est clairement semi-décidable. Donc $\complement\mathscr{H}$ n'est pas semi-décidable.

(d) Supposons par l'absurde qu'on ait $\hat{F}(e_d) \neq F(g)$ pour tout $d \in \mathscr{H}$: comme on a vu en (a) que $\hat{F}(e_d) = F(g)$ pour tout $d \notin \mathscr{H}$, on a alors $\{d \in \mathbb{N} : \hat{F}(e_d) \neq F(g)\} = \mathscr{H}$, autrement dit $\{d \in \mathbb{N} : \hat{F}(e_d) = F(g)\} = \complement\mathscr{H}$, or on a vu en (b) que $\{d \in \mathbb{N} : \hat{F}(e_d) = F(g)\}$ est semi-décidable et en (c) que $\complement\mathscr{H}$ ne l'est pas, une contradiction. Donc il doit exister $d \in \mathscr{H}$ tel que $\hat{F}(e_d) = F(g)$. ✓

(7) On a montré en (6) qu'il existe d tel que $\varphi_d(0)\!\downarrow$ et que $\hat{F}(e_d) = F(g)$. Soit n le nombre d'étapes d'exécution[5] de $\varphi_d(0)$. Montrer que pour tout $g' \in \mathcal{B}_0(g,n)$ on a $F(g') = F(g)$. (*Indication :* sinon, la recherche d'un j dans l'algorithme A aurait réussi à trouver un j : on pourra montrer qu'on aurait $g_d = \varphi_{\gamma(g,n,j)}$ donc $F(g_d) \neq F(g)$ dans ce cas.)

Corrigé. Supposons par l'absurde que $F(g') \neq F(g)$ pour un certain $g' \in \mathcal{B}_0(g,n)$. Comme on l'a vu dans la question (4), on a $g' = \varphi_{\gamma(g,n,j_0)}$ pour un certain j_0, et en particulier, $F(g') = F(\varphi_{\gamma(g,n,j_0)}) = \hat{F}(\gamma(g,n,j_0))$ donc $\hat{F}(\gamma(g,n,j_0)) \neq F(g)$. Dans l'exécution de l'algorithme A, si $\ell \geq n$, on est dans le cas où l'exécution de $\varphi_d(0)$ s'est terminée dans le temps imparti (par définition de n), et la seconde boucle va trouver un j tel que $\hat{F}(\gamma(g,n,j)) \neq F(g)$ (puisqu'il en existe un),

5. Ou si on préfère : le code de l'arbre de calcul.

pas forcément j_0 mais toujours le même quel que soit ℓ, appelons-le j_1, et l'algorithme va renvoyer $\varphi_{\gamma(g,n,j_1)}(\ell)$; tandis que si $\ell < n$, alors l'exécution de $\varphi_d(0)$ ne sera terminée dans le temps imparti et l'algorithme va renvoyer $g(\ell)$, qui est aussi $\varphi_{\gamma(g,n,j_1)}(\ell)$ puisque $\gamma(g,n,j_1) \in \mathcal{B}(g,n)$. Donc dans tous les cas, l'algorithme renvoie $\varphi_{\gamma(g,n,j_1)}(\ell)$, c'est-à-dire que $g_d = \varphi_{\gamma(g,n,j_1)}$. Or la définition de j_1 fait que $\hat{F}(\gamma(g,n,j_1)) \neq F(g)$, et on a $F(g_d) = F(\varphi_{\gamma(g,n,j_1)}) = \hat{F}(\gamma(g,n,j_1))$, ce qui mis ensemble montre que $F(g_d) \neq F(g)$. Mais ceci contredit le fait que $F(g_d) = \hat{F}(e_d) = F(g)$. On a bien abouti à une contradiction. C'est donc que $F(g') = F(g)$ pour tout $g' \in \mathcal{B}_0(g,n)$. ✓

(8) On a montré en (7) que pour tout $g \in \mathsf{TotCalc}$ il existe n tel que pour tout $g' \in \mathcal{B}_0(g,n)$ on a $F(g') = F(g)$. On considère maintenant $g' \in \mathcal{B}(g,n)$ (qui n'est plus supposé nul à partir d'un certain rang) : d'après ce qu'on vient de dire, il existe n' tel que pour tout $g'' \in \mathcal{B}_0(g',n')$ on a $F(g'') = F(g')$. En justifiant qu'on peut trouver $g'' \in \mathcal{B}_0(g,n) \cap \mathcal{B}_0(g',n')$, montrer que $F(g') = F(g)$. Conclure.

Corrigé. Renommons en g'' le g' de la question (7) pour y voir plus clair : on a trouvé un n tel que pour tout $g'' \in \mathcal{B}_0(g,n)$ on a $F(g'') = F(g)$. Si maintenant $g' \in \mathcal{B}(g,n)$, on trouve de la même manière n' tel que pour tout $g'' \in \mathcal{B}_0(g',n')$ on a $F(g'') = F(g')$. Si $n'' = \max(n,n')$, alors n'importe quel $g'' \in \mathcal{B}_0(g',n'')$ coïncide avec g' jusqu'au rang n'' donc jusqu'au rang n', et par ailleurs avec g' jusqu'au rang n'' donc jusqu'au rang n donc aussi avec g jusqu'au rang n : ceci montre $g'' \in \mathcal{B}_0(g,n) \cap \mathcal{B}_0(g',n')$ (en fait, $\mathcal{B}_0(g',n'') \subseteq \mathcal{B}_0(g,n) \cap \mathcal{B}_0(g',n')$), et on a $F(g'') = F(g)$ puisque $g'' \in \mathcal{B}_0(g,n)$ et $F(g'') = F(g')$ puisque $g'' \in \mathcal{B}_0(g',n')$, donc $F(g') = F(g'') = F(g)$.

On a donc bien montré l'existence d'un certain n tel que $F(g') = F(g)$ pour tout $g' \in \mathcal{B}(g,n)$. ✓

(9) En observant que l'algorithme A peut s'écrire (à g variable) à partir du code d'un programme calculant g, justifier qu'un n dont on a montré l'existence en (8) peut être obtenu de façon calculable en fonction du code d'un programme calculant g. (On esquissera un algorithme B qui calcule un n en fonction d'un code r d'un programme calculant g.)

Corrigé. Observons d'abord que l'algorithme A peut s'écrire (i.e., « être rendu uniforme ») à partir du code r d'un programme calculant $g = \varphi_r$: on a déjà observé en (4) que c'est le cas de $\gamma(g,n,j)$, et dans l'algorithme A, on peut remplacer l'invocation de $g(\ell)$ par $\varphi_r(\ell)$ (calculable au moyen d'un programme universel), et celle de $F(g)$ par $\hat{F}(r)$. Au moyen du théorème s-m-n (comme on l'a déjà expliqué plusieurs fois ci-dessus), on en déduit une fonction calculable $(r,d) \mapsto e_{r,d}$ qui (si $r \in \mathsf{TotCode}$) renvoie le code d'un $e_{r,d}$ calculé par l'algorithme en question (i.e., un code pour g_d si $g = \varphi_r$).

Reste maintenant expliquer comment trouver n en fonction de ce r. Pour cela, l'algorithme B sera le suivant. On parcourt tous les couples (d,n) (disons, au moyen du code de Gödel $\langle d,n \rangle$ du couple), et, si $\varphi_d(0)$ termine en exactement n étapes, on teste si $\hat{F}(e_{r,d}) = \hat{F}(r)$ (où $e_{r,d}$ est, comme on vient de le dire, essentiellement la curryfication de l'algorithme A, et $\hat{F}$ est donné). Lorsqu'un (d,n) est trouvé, on renvoie le n en question.

D'après la question (6) (et le fait que $\hat{F}(r) = F(g)$), il existe bien un tel (d,n) comme recherché par l'algorithme B, donc celui-ci termine. D'après les question (7) et (8), le n qu'il renvoie vérifie la propriété demandée. ✓

(10) On a montré en (3)–(8) le théorème de Kreisel-Lacombe-Shoenfield qui affirme que

$$\forall F : \mathsf{TotCalc} \to \mathbb{N} \text{ calculable. } \forall g \in \mathsf{TotCalc}. \ \exists n \in \mathbb{N}. \ \forall g' \in \mathcal{B}(g,n). \ (F(g') = F(g))$$

Montrer par un exemple simple que l'affirmation suivante [6]

$$\forall F : \mathsf{TotCalc} \to \mathbb{N} \text{ calculable. } \exists n \in \mathbb{N}. \ \forall g \in \mathsf{TotCalc}. \ \forall g' \in \mathcal{B}(g,n). \ (F(g') = F(g))$$

6. Ce serait une affirmation de continuité *uniforme* de F.

n'est pas valable. (*Indication :* proposer une fonction F qui évalue un nombre de valeurs $g(i)$ de g dépendant, disons, de $g(0)$.)

Corrigé. On peut par exemple considérer la fonction(nelle) F suivante : calculer $g(0) =: N$, puis calculer et renvoyer la somme des valeurs $g(1) + g(2) + \cdots + g(N)$. Manifestement, F est bien calculable. Supposons par l'absurde qu'il existe n tel que $\forall g \in \mathsf{TotCalc}.\ \forall g' \in \mathcal{B}(g,n).\ (F(g') = F(g))$. Considérons la fonction (évidemment calculable) qui vaut $n+1$ en 0 et 0 en tout $i > 0$: ainsi, $F(g) = 0$, donc on devrait avoir $F(g')$ pour tout $g' \in \mathcal{B}(g,n)$, c'est-à-dire tout g' tel que $g'(0) = n+1$ et $g'(1) = \cdots = g'(n) = 0$. Or en prenant g' valant 42 en $n+1$ et toujours 0 ensuite, on a $F(g') = 42$, donc $F(g') \neq F(g)$, ce qui contredit l'affirmation. ✓

(11) On a prouvé en (9) qu'un n (tel que $\forall g' \in \mathcal{B}(g,n).\ (F(g') = F(g))$) peut être calculé algorithmiquement en fonction du code r d'un programme qui calcule g. En fait, on pourrait prouver un peu mieux (*ceci n'est pas demandé*) : il est possible de calculer un tel n par un algorithme qui a seulement le droit d'interroger les valeurs [7] de la fonction g. En *admettant* ce fait, expliquer pourquoi la fonction F elle-même peut être calculée par un tel algorithme.

Corrigé. Pour la complétude de ce corrigé, et *bien que ce ne soit pas demandé par le sujet*, prouvons l'affirmation admise. Appelons $N(r)$ la valeur calculée par l'algorithme B donné dans la question (9), c'est-à-dire en supposant qu'on ait accès au code r du programme qui calcule g. On cherche maintenant à faire pareil sans avoir accès à ce code. Pour cela, considérons l'algorithme C suivant : on recherche un $r \in \mathbb{N}$ vérifiant les conditions :

— $N(r)\!\downarrow$,
— $\varphi_r(i) = g(i)$ pour tout $i \leq N(r)$,
— $\hat{F}(\gamma(g, N(r), 0)) = \hat{F}(r)$;

et une fois qu'il est trouvé, on renvoie $N(r)$.

Un tel r existe bien car si r_0 est tel que $\varphi_{r_0} = g$, alors ce r_0 vérifie bien les trois conditions (la première vient de $r_0 \in \mathsf{TotCode}$, la deuxième est une conséquence triviale de $\varphi_{r_0} = g$, et la troisième découle du fait que tout $g' \in \mathcal{B}(g, N(r_0))$ vérifie $F(g') = F(g)$, et que $g' := \varphi_{\gamma(g,N(r_0),0)} \in \mathcal{B}_0(g, N(r_0))$ donc $\hat{F}(\gamma(g, N(r), 0)) = F(g') = F(g) = \hat{F}(r_0)$).

La recherche d'un r est possible algorithmiquement bien que les trois contraintes ne soient que semi-décidables (le calcul de $\varphi_r(i)$ pourrait ne pas terminer, comme celui de $N(r)$ ou de $\hat{F}(r)$, puisque rien ne garantit que $r \in \mathsf{TotCode}$) : en effet, on peut à lancer en parallèle toutes les recherches (c'est-à-dire qu'on parcourt les couples (r, M) et pour chacun on fait M étapes de la vérification que r convient, jusqu'à trouver un r qui vérifie les contraintes).

L'algorithme C qu'on vient de dire n'utilise que l'interrogation de valeurs de g (c'est trivial pour le premier point qui ne fait pas intervenir g, pour le deuxième le test se fait en utilisant les valeurs $g(0), \ldots, g(N(r))$, et pour le troisième, il est clair dans la construction faite en (4) que pour déterminer $\gamma(g, n, j)$ on n'a besoin que d'interroger les valeurs de g, pas d'avoir accès à son code).

Une fois qu'un tel r est trouvé, même sans supposer que la fonction φ_r soit totale, l'argument donné en (7) montre qu'on ne peut pas y avoir de j tel que $\hat{F}(\gamma(g, N(r), j)) \neq \hat{F}(r)$, donc comme en (8), on a $F(g') = \hat{F}(r)$ pour tout $g' \in \mathcal{B}(g, N(r))$, et $N(r)$ vérifie bien la contrainte demandée.

Ceci conclut la preuve du fait qui était admis. Passons maintenant à la question proprement dite.

Pour calculer $F(g)$ à partir de l'interrogation de valeurs de g, on commence par calculer, en interrogeant les valeurs de g, un n tel que $\forall g' \in \mathcal{B}(g,n).\ (F(g') = F(g))$ (on vient d'admettre ou d'expliquer qu'on peut le faire). Puis on calcule et renvoie $\hat{F}(\gamma(g, n, 0))$, ce qui peut se faire

7. C'est-à-dire : un algorithme qui a accès à un oracle calculant qui renvoie $g(i)$ quand on lui soumet la valeur i. L'algorithme a le droit d'appeler librement l'oracle un nombre fini quelconque de fois au cours de son exécution.

en n'interrogeant qu'un nombre fini de valeurs de g (à savoir $g(0), \ldots, g(n)$) : comme $\gamma(g, n, 0) \in \mathcal{B}(g, n)$ par définition de γ, on a bien $\hat{F}(\gamma(g, n, 0)) = F(g)$, et on a calculé cette valeur en passant simplement par l'interrogation de valeurs de g et sans avoir besoin de connaître son code. ✓

(12) Quelle est la morale de l'ensemble de ce problème ? Autrement dit, expliquer *en français de façon informelle* le sens du théorème de Kreisel-Lacombe-Shoenfield : on cherchera à dire quelque chose de la forme « la seule manière de construire une fonction totale calculable sur l'ensemble des fonctions est la suivante ». On contrastera avec la situation des questions (1) et (2).

Corrigé. On s'est interrogé dans ce problème sur ce que peuvent faire des algorithmes qui prennent en entrée un autre programme et qui calculent un entier. Dans la question (1), on a vu, comme variation sur le théorème de Rice, que si le programme fourni en entrée de l'algorithme n'est pas supposé terminer à coup sûr, alors on ne peut rien faire d'intelligent : on ne peut que renvoyer toujours la même valeur.

En revanche, lorsque l'algorithme fourni en entrée est supposé toujours terminer, on peut au moins évaluer ses valeurs (c'est ce qu'on a fait dans la question (2)), c'est-à-dire qu'on peut renvoyer des expressions faisant intervenir les $g(i)$. Ce qu'affirme le théorème de Kreisel-Lacombe-Shoenfield, c'est que toute valeur $F(g)$ calculée ne dépend, en fait, que d'un nombre fini de telles valeurs $g(0), \ldots, g(n)$, et, finalement (c'est la question (11)) peut être calculé par un algorithme qui n'a le droit d'accéder qu'à une boîte noire renvoyant la valeur de $g(i)$ en fonction de i.

La question (10) montre que, si pour n'importe quel g le nombre de valeurs dont dépend $F(g)$ est borné, il n'est pas forcément *uniformément borné*, c'est-à-dire qu'on ne peut pas forcément trouver un seul et même n tel que la connaissance des n premières valeurs de g suffise à calculer $F(g)$. ✓