INF110
Contrôle de connaissances — Corrigé

Logique et Fondements de l'Informatique

26 janvier 2024

Consignes.

Les exercices et le problème sont totalement indépendants. 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 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.

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

Barème *approximatif* et *indicatif* (sur 20) : entre 2 et 3 points par exercice et environ 0.75 points par question du problème.

Ce corrigé comporte 13 pages (page de garde incluse).

Git : 334c718 Wed Jan 24 23:11:29 2024 +0100

Exercice 1.

On considère un extrait d'une démonstration interactive en Coq du théorème `forall (A:Prop) (B:Prop) (C:Prop), (A->B)->(B->C)->(A->C)` où on a recopié ci-dessous seulement la sortie de Coq. On demande de retrouver les deux commandes qui ont été utilisées, et de proposer l'étape suivante.

```
1 subgoal

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

`thm <` (commande à trouver)

```
1 subgoal

A, B, C : Prop
x : A -> B
y : B -> C
z : A
============================
C
```

`thm <` (commande à trouver)

```
1 subgoal

A, B, C : Prop
x : A -> B
y : B -> C
z : A
============================
B
```

`thm <` (commande à proposer)

(À défaut de connaître le nom précis de la tactique, on pourra expliquer ce qu'elle fait de façon générale, ou à quelle règle de logique elle correspond.)

Corrigé. La première commande était `intros x y z.` et correspond à la règle d'introduction du $\Rightarrow$ appliquée trois fois. La commande suivante était `apply y.` et correspond à la règle d'élimination du $\Rightarrow$ entre l'hypothèse (`y`) qu'on lui indique et la démonstration qu'onva produire du nouveau but. Une commande naturelle ensuite serait `apply x.` (qui ramènerait le but à `A`, et qui pourrait être suivie de `exact z.` pour terminer la démonstration). ✓

Exercice 2.

Donner une démonstration en calcul propositionnel intuitionniste de la formule suivante :

$$A \Rightarrow (A \Rightarrow B) \Rightarrow B$$

On donnera la démonstration sous la forme qu'on préfère (arbre de preuve ou style drapeau), puis on donnera aussi un λ-terme de preuve. On décrira ensuite brièvement ce que fait ce terme vu en tant

que programme (i.e., le comportement du programme associé à la preuve par la correspondance de Curry-Howard).

Corrigé. Voici une preuve complète écrite dans le style « drapeau » :

(1) $\boxed{A}$

(2) $\quad\boxed{A \Rightarrow B}$

(3) $\quad B$ $\qquad\qquad$ $\Rightarrow$Élim sur (2) et (1)

(4) $\quad (A \Rightarrow B) \Rightarrow B$ $\qquad$ $\Rightarrow$Int de (2) dans (3)

(5) $A \Rightarrow (A \Rightarrow B) \Rightarrow B$ $\qquad$ $\Rightarrow$Int de (2) dans (4)

La voici écrite sous forme d'arbre de preuve :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A,\ A \Rightarrow B \vdash A \Rightarrow B}\ \textsc{Ax} \qquad \dfrac{}{A,\ A \Rightarrow B \vdash A}\ \textsc{Ax}}{A, A \Rightarrow B \vdash B}\ \Rightarrow\textsc{Élim}}{A \vdash (A \Rightarrow B) \Rightarrow B}\ \Rightarrow\textsc{Int}}{\vdash A \Rightarrow (A \Rightarrow B) \Rightarrow B}\ \Rightarrow\textsc{Int}$$

ou, si on veut donner des noms aux hypothèses et marquer tous les termes de preuve intermédiaires :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : A,\ k : A \Rightarrow B\ \vdash\ k : A \Rightarrow B}\ \textsc{Ax} \qquad \dfrac{}{x : A,\ k : A \Rightarrow B\ \vdash\ x : A}\ \textsc{Ax}}{x : A,\ k : A \Rightarrow B\ \vdash\ kx : B}\ \Rightarrow\textsc{Élim}}{x : A\ \vdash\ \lambda(k : A \Rightarrow B).\, kx\ :\ (A \Rightarrow B) \Rightarrow B}\ \Rightarrow\textsc{Int}}{\vdash\ \lambda(x : A).\, \lambda(k : A \Rightarrow B).\, kx\ :\ A \Rightarrow (A \Rightarrow B) \Rightarrow B}\ \Rightarrow\textsc{Int}$$

Le λ-terme est donc $\lambda(x : A).\, \lambda(k : A \Rightarrow B).\, kx$, et le programme transforme x en la fonction qui applique une fonction k à x, ou, ce qui revient au même, il prend successivement x et k et applique k à x. ✓

Exercice 3.

(Dans cet exercice, on ne demande pas de justifier les réponses.)

(1) En calcul propositionnel intuitionniste, quel est l'énoncé prouvé par le λ-terme de preuve suivant ? (Ou si on préfère : quel est le type de ce terme du λ-calcul simplement typé enrichi de types produits, qu'on a écrit en notations logiques ?)

$$\lambda(p : C \wedge (A \Rightarrow B)).\, \lambda(h : A).\, \langle(\pi_1 p),\ (\pi_2 p)h\rangle$$

(Ici, A, B, C sont des variables propositionnelles.)

(2) En logique du premier ordre intuitionniste, quel est l'énoncé prouvé par le λ-terme de preuve suivant ?

$$\lambda(p : C \wedge \forall x.B(x)).\, \lambda(x : I).\, \langle(\pi_1 p),\ (\pi_2 p)x\rangle$$

(Ici, C est une variable de relation nullaire, c'est-à-dire une variable propositionnelle, B est une variable de relation unaire, et I est le symbole du type des individus.)

(3) En logique du premier ordre intuitionniste, quel est l'énoncé prouvé par le λ-terme de preuve suivant ?

$$\lambda(p : \exists x.(A \Rightarrow B(x))).\, \lambda(h : A).\, (\texttt{match}\ p\ \texttt{with}\ \langle x, j\rangle \mapsto \langle x, jh\rangle)$$

(Ici, A est une variable de relation nullaire, c'est-à-dire une variable propositionnelle, et B est une variable de relation unaire.)

Corrigé. **(1)** Dans le contexte où p a type $C \wedge (A \Rightarrow B)$ et h a type A, on voit que $\pi_1 p$ a type C et $\pi_2 p$ a type $A \Rightarrow B$, si bien que $(\pi_2 p)h$ a type B, et le couple $\langle(\pi_1 p),\ (\pi_2 p)h\rangle$ a type $C \wedge B$, donc l'expression dans son ensemble a type $C \wedge (A \Rightarrow B) \Rightarrow (A \Rightarrow C \wedge B)$, i.e., elle représente une preuve de cette affirmation.

(2) On se place dans le contexte où p a type $C \wedge \forall z.B(z)$ et x est un individu. Alors $\pi_1 p$ a type C et $\pi_2 p$ a type $\forall z.B(z)$, si bien que $(\pi_2 p)x$ a type $B(x)$, et $\langle(\pi_1 p),\ (\pi_2 p)x\rangle$ a type $C \wedge B(x)$, donc l'expression dans son ensemble a type $(C \wedge \forall x.B(x)) \Rightarrow \forall x.(C \wedge B(x))$, i.e., elle représente une preuve de cette affirmation.

(3) On se place dans le contexte où p a type $\exists x.(A \Rightarrow B(x))$ et h a type A. Le `match` va déstructurer p en un x individu et un j de type $A \Rightarrow B(x)$. Alors jh a type $B(x)$, et $\langle x, jh\rangle$ a type $\exists x.B(x)$. Donc l'expression dans son ensemble a type $(\exists x.(A \Rightarrow B(x))) \Rightarrow (A \Rightarrow \exists x.B(x))$, i.e., elle représente une preuve de cette affirmation. ✓

Exercice 4.

En appliquant l'algorithme de Hindley-Milner, trouver une annotation de type (dans le λ-calcul simplement typé) au terme suivant du λ-calcul non typé :

$$\lambda f.\, \lambda z.\, f z(\lambda g.gz)$$

(autrement dit, `fun f -> fun z -> f z (fun g -> g z)` en syntaxe OCaml). Quel théorème du calcul propositionnel intuitionniste obtient-on de ceci par Curry-Howard ?

(Une réponse qui ne suit pas l'algorithme de Hindley-Milner mais qui est néanmoins correcte vaudra une partie des points.)

Corrigé. Dans une première phase, on collecte les types et contraintes suivants : $f : \eta_1$, $z : \eta_2$, $fz : \eta_3$ avec $\eta_1 = (\eta_2 \to \eta_3)$, $g : \eta_4$, $gz : \eta_5$ avec $\eta_4 = (\eta_2 \to \eta_5)$ donc $\lambda g.gz : \eta_4 \to \eta_5$ et enfin $fz(\lambda g.gz) : \eta_6$ avec $\eta_3 = ((\eta_4 \to \eta_5) \to \eta_6)$; et le type final est $\eta_1 \to \eta_2 \to \eta_6$. On a donc les contraintes suivantes :

$$\begin{aligned}\eta_1 &= (\eta_2 \to \eta_3)\\ \eta_4 &= (\eta_2 \to \eta_5)\\ \eta_3 &= ((\eta_4 \to \eta_5) \to \eta_6)\end{aligned}$$

On examine d'abord la contrainte $\eta_1 = (\eta_2 \to \eta_3)$: η_1 est une variable de type, et elle n'apparaît pas dans l'autre membre de la contrainte ; il n'y a rien à faire à part enregistrer $\eta_1 \mapsto (\eta_2 \to \eta_3)$ dans la substitution. On examine ensuite la contrainte $\eta_4 = (\eta_2 \to \eta_5)$: on enregistre $\eta_4 \mapsto (\eta_2 \to \eta_5)$ dans la substitution, et on l'applique à la dernière contrainte, qui devient $\eta_3 = (((\eta_2 \to \eta_5) \to \eta_5) \to \eta_6)$. Enfin, on examine la contrainte $\eta_3 = (((\eta_2 \to \eta_5) \to \eta_5) \to \eta_6)$: on enregistre $\eta_3 \mapsto (((\eta_2 \to \eta_5) \to \eta_5) \to \eta_6)$ dans la substitution et on l'applique à $\eta_1 \mapsto (\eta_2 \to \eta_3)$, qui devient $\eta_1 \mapsto (\eta_2 \to ((\eta_2 \to \eta_5) \to \eta_5) \to \eta_6)$. Finalement, on a trouvé la substitution :

$$\begin{aligned}&\eta_1 \mapsto (\eta_2 \to ((\eta_2 \to \eta_5) \to \eta_5) \to \eta_6)\\ &\eta_4 \mapsto (\eta_2 \to \eta_5)\\ &\eta_3 \mapsto (((\eta_2 \to \eta_5) \to \eta_5) \to \eta_6)\end{aligned}$$

qu'on applique à tous les types intermédiaires et au type final, ce qui donne

$$\begin{aligned}&\lambda(f : \eta_2 \to ((\eta_2 \to \eta_5) \to \eta_5) \to \eta_6).\, \lambda(z : \eta_2).\, fz(\lambda(g : \eta_2 \to \eta_5).gz)\\ :\quad &(\eta_2 \to ((\eta_2 \to \eta_5) \to \eta_5) \to \eta_6) \to \eta_2 \to \eta_6\end{aligned}$$

En particulier, on voit que $(H_2 \Rightarrow ((H_2 \Rightarrow H_5) \Rightarrow H_5) \Rightarrow H_6) \Rightarrow H_2 \Rightarrow H_6$ est un théorème du calcul propositionnel intuitionniste. ✓

Exercice 5.

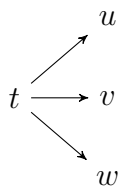
En utilisant l'une quelconque des sémantiques vues en cours pour le calcul propositionnel intuitionniste (au choix : toutes marcheront), montrer que la formule suivante n'est pas démontrable en calcul propositionnel intuitionniste :

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

(*Indications.* Si on préfère la sémantique de Kripke, considérer un monde permettant d'accéder à trois autres mondes, tous les trois inaccessibles depuis les uns les autres. Si on préfère la sémantique des ouverts, considérer trois secteurs angulaires ouverts dans le plan, ayant un même sommet et qui se jouxtent sans s'intersecter. Si on préfère la sémantique de la réalisabilité ou des problèmes finis, il s'agit d'énoncer le fait qu'il est impossible de déterminer une partie vide parmi trois, sans avoir accès à ces parties, même si on a la promesse qu'au moins deux d'entre elles sont vides : pour la réalisabilité, on pourra considérer une des parties valant $\mathbb{N}$ et les deux autres valant $\varnothing$ et constater qu'on ne peut pas réaliser les trois affectations à la fois, tandis que pour la sémantique des problèmes finis, on pourra considérer trois problèmes ayant un seul candidat chacun mais un seul ayant une solution.)

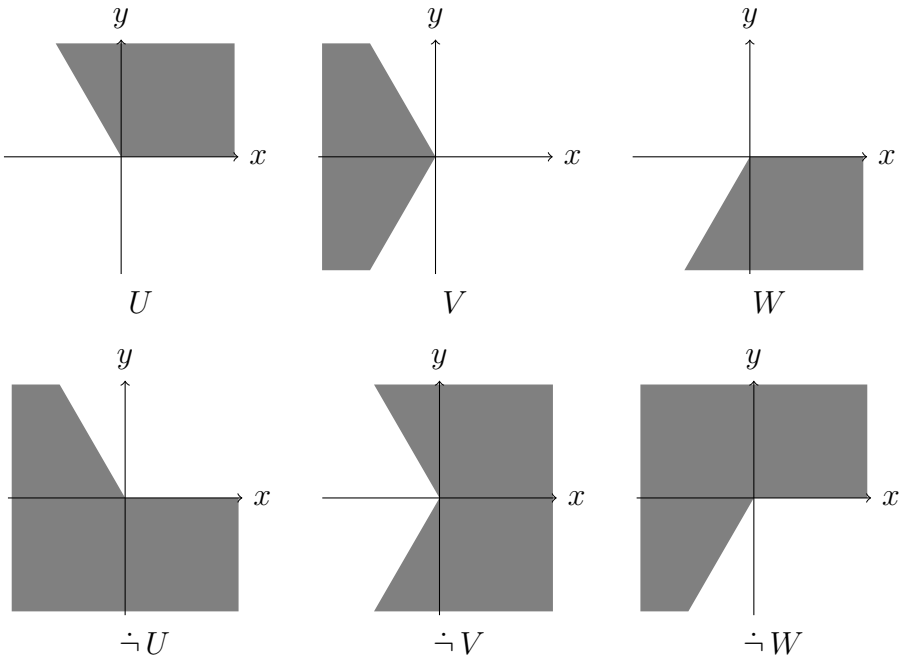
Corrigé. On donne quatre solutions possibles, une pour chaque sémantique vue en cours.

Sémantique de Kripke : On considère le cadre de Kripke suivant :



Soient p (resp. q, resp. r) l'affectation de vérité qui vaut 1 en u (resp. v, resp. w) et 0 partout ailleurs. Alors chacun de $p \mathbin{\dot\wedge} q$, $q \mathbin{\dot\wedge} r$ et $r \mathbin{\dot\wedge} p$ est identiquement 0 donc $\dot\neg(p \mathbin{\dot\wedge} q) \mathbin{\dot\wedge} \dot\neg(q \mathbin{\dot\wedge} r) \mathbin{\dot\wedge} \dot\neg(r \mathbin{\dot\wedge} p)$ est identiquement 1. En revanche, $\dot\neg\, p$ vaut 1 dans les mondes v et w et 0 dans les mondes t et u, et de façon analogue pour $\dot\neg\, q$ et $\dot\neg\, r$ avec la permutation cyclique évidente. Donc $\dot\neg\, p \mathbin{\dot\vee} \dot\neg\, q \mathbin{\dot\vee} \dot\neg\, r$ vaut 0 dans le monde t et 1 dans les trois mondes u, v, w. Par conséquent, il en va de même de $(\dot\neg(p \mathbin{\dot\wedge} q) \mathbin{\dot\wedge} \dot\neg(q \mathbin{\dot\wedge} r) \mathbin{\dot\wedge} \dot\neg(r \mathbin{\dot\wedge} p)) \mathbin{\dot\Rightarrow} (\dot\neg\, p \mathbin{\dot\vee} \dot\neg\, q \mathbin{\dot\vee} \dot\neg\, r)$. Si la formule proposée avait été démontrable, on aurait trouvé constamment 1 (par *correction* de la sémantique de Kripke) : ce n'est pas le cas, donc la formule n'est pas démontrable.

Sémantique des ouverts : Considérons dans $\mathbb{R}^2$ trois secteurs angulaires ouverts U, V, W, chacun ayant l'origine comme sommet et d'angle $\frac{2\pi}{3}$, et qui se jouxtent deux par deux : par exemple, soit U le secteur formé des nombres complexes non nuls dont l'argument (choisi entre 0 et 2π) est strictement compris entre 0 et $\frac{2\pi}{3}$, soit V celui dont l'argument est strictement compris entre $\frac{2\pi}{3}$ et $\frac{4\pi}{3}$ et soit W celui dont l'argument est strictement compris entre $\frac{4\pi}{3}$ et 2π. Avec ces choix, les trois secteurs sont deux à deux disjoints, donc chacun de $U \mathbin{\dot\wedge} V$, $V \mathbin{\dot\wedge} W$ et $W \mathbin{\dot\wedge} U$ est vide donc $\dot\neg(U \mathbin{\dot\wedge} V) \mathbin{\dot\wedge} \dot\neg(V \mathbin{\dot\wedge} W) \mathbin{\dot\wedge} \dot\neg(W \mathbin{\dot\wedge} U)$ est plein. En revanche, $\dot\neg\, U$, $\dot\neg\, V$ et $\dot\neg\, W$ sont des secteurs ouverts ayant l'origine pour sommet et d'angle $\frac{4\pi}{3}$ (par exemple, $\dot\neg\, U$ est l'intérieur de l'adhérence de $V \cup W$, et symétriquement pour les autres), donc aucun ne contient l'origine, donc $\dot\neg\, U \mathbin{\dot\vee} \dot\neg\, V \mathbin{\dot\vee} \dot\neg\, W$ non plus : c'est même précisément le complémentaire de l'origine. Par conséquent, $(\dot\neg(U \mathbin{\dot\wedge} V) \mathbin{\dot\wedge} \dot\neg(V \mathbin{\dot\wedge} W) \mathbin{\dot\wedge} \dot\neg(W \mathbin{\dot\wedge} U)) \mathbin{\dot\Rightarrow} (\dot\neg\, U \mathbin{\dot\vee} \dot\neg\, V \mathbin{\dot\vee} \dot\neg\, W)$ est aussi le complémentaire de l'origine. Si la formule proposée avait été démontrable, on aurait trouvé tout $\mathbb{R}^2$ (par *correction* de la sémantique des ouverts) : ce n'est pas le cas, donc la formule n'est pas démontrable.



Sémantique de la réalisabilité : Supposons qu'il existe un *même* programme e qui réalise la formule qu'on considère quelles que soient les parties $P, Q, R \subseteq \mathbb{N}$ mises à la place des variables propositionnelles A, B, C. Notons que si (au moins) deux des trois parties P, Q, R sont vides, alors $\dot{\neg}(P \dot{\wedge} Q)$, $\dot{\neg}(Q \dot{\wedge} R)$ et $\dot{\neg}(R \dot{\wedge} P)$ sont égales à $\mathbb{N}$, donc $\langle 0, 0, 0 \rangle$ est dans $\dot{\neg}(P \dot{\wedge} Q) \dot{\wedge} \dot{\neg}(Q \dot{\wedge} R) \dot{\wedge} \dot{\neg}(R \dot{\wedge} P)$. Considérons $\varphi_e(\langle 0, 0, 0 \rangle)$: il est bien défini (car il existe des parties P, Q, R comme on l'a dit) et doit appartenir à $\dot{\neg} P \,\dot{\vee}\, \dot{\neg} Q \,\dot{\vee}\, \dot{\neg} R$, c'est-à-dire qu'il doit indiquer une partie vide parmi P, Q, R. Mais ce n'est pas possible sans aucune information sur les parties ! Pour être tout à fait précis, supposons (sans perte de généralité, puisque le problème est symétrique) qu'il renvoie un élément de $\dot{\neg} P$: alors en considérant le cas $P = \mathbb{N}$ et $Q = \varnothing$ et $R = \varnothing$ on a une contradiction. Or si la formule proposée avait été démontrable, il existerait un e qui la réalise quelles que soient les parties (par *correction* de la sémantique des la réalisabilité) : on vient de voir que ce n'est pas le cas, donc la formule n'est pas démontrable.

Sémantique des problèmes finis : Considérons trois problèmes dont l'ensemble des candidats est un singleton, appelons-les $E := (\{\bullet_p\}, P)$, $F := (\{\bullet_q\}, Q)$ et $G := (\{\bullet_r\}, R)$ où chacun des ensembles de solutions P, Q, R est soit vide soit le singleton écrit juste avant. Alors l'ensemble des candidats de $\dot{\neg}(E \,\dot{\wedge}\, F) \,\dot{\wedge}\, \dot{\neg}(F \,\dot{\wedge}\, G) \,\dot{\wedge}\, \dot{\neg}(G \,\dot{\wedge}\, E)$ est un singleton, appelons-le $\{\bullet_0\}$, et ce singleton est solution lorsque (au moins) deux des trois parties P, Q, R sont vides. L'ensemble des candidats de chacun de $\dot{\neg} E$, $\dot{\neg} F$, $\dot{\neg} G$ est un singleton, disons $\{\bullet_{p'}\}$, $\{\bullet_{q'}\}$, $\{\bullet_{r'}\}$ respectivement, donc celui de $\dot{\neg} E \,\dot{\vee}\, \dot{\neg} F \,\dot{\vee}\, \dot{\neg} G$ est $\{\bullet_{p'}, \bullet_{q'}, \bullet_{r'}\}$. Supposons qu'il existe un *même* candidat f qui soit solution du problème $(\dot{\neg}(E \,\dot{\wedge}\, F) \,\dot{\wedge}\, \dot{\neg}(F \,\dot{\wedge}\, G) \,\dot{\wedge}\, \dot{\neg}(G \,\dot{\wedge}\, E)) \mathrel{\dot{\Rightarrow}} (\dot{\neg} E \,\dot{\vee}\, \dot{\neg} F \,\dot{\vee}\, \dot{\neg} G)$. C'est une fonction $\{\bullet_0\} \to \{\bullet_{p'}, \bullet_{q'}, \bullet_{r'}\}$ censée envoyer $\bullet_0$, s'il est solution, sur une solution du problème $\dot{\neg} E \,\dot{\vee}\, \dot{\neg} F \,\dot{\vee}\, \dot{\neg} G$, c'est-à-dire qu'il doit indiquer une partie vide parmi P, Q, R. Mais ce n'est pas possible sans aucune information sur les parties ! Pour être tout à fait précis, supposons (sans perte de généralité, puisque le problème est symétrique) que $f(\bullet_0) = \bullet_{p'}$: alors en considérant le cas $P = \{\bullet_p\}$ et $Q = \varnothing$ et $R = \varnothing$ on a une contradiction. Or si la formule proposée avait été démontrable, il existerait un f qui soit solution quels que soient P, Q, R (par *correction* de la sémantique des la réalisabilité) : on vient de voir que ce n'est pas le cas, donc la formule n'est pas démontrable. ✓

Exercice 6.

(On ne demande pas ici de réponses compliquées !)

(1) Expliquer rapidement pourquoi, si $\mathscr{T}$ est un ensemble de formules logiques (par exemple de logique de premier ordre, mais peu importent les détails), on peut démontrer $P \Rightarrow Q$ à partir de $\mathscr{T}$ si et seulement si on peut démontrer Q à partir de $\mathscr{T} \cup \{P\}$.

(2) Déduire de (1) et du théorème de Gödel que la théorie PA^* formée en ajoutant l'axiome $\neg\mathsf{Consis}(\mathsf{PA})$ à l'arithmétique de Peano (PA), ne prouve pas $\bot$ (c'est-à-dire qu'elle n'est pas contradictoire). Ici, $\mathsf{Consis}(\mathsf{PA})$ est l'énoncé qui affirme que PA n'est pas contradictoire, et on rappelle que cet énoncé est démontrable dans les mathématiques usuelles (ZFC) où on travaille.

(On rappelle aussi que PA travaille en logique classique.)

(3) Expliquer pourquoi $\mathsf{Consis}(\mathsf{PA}^*)$ est plus fort que (i.e., implique) $\mathsf{Consis}(\mathsf{PA})$, et en déduire que PA^* démontre $\neg\mathsf{Consis}(\mathsf{PA}^*)$.

(4) On a vu en (2) que PA^* n'est pas contradictoire, et en (3) que PA^* démontre que PA^* est contradictoire. Ceci peut sembler paradoxal. Qu'en pensez-vous ?

Corrigé. **(1)** Les règles d'introduction et d'élimination de $\Rightarrow$ qui affirment que $\Gamma, P \vdash Q$ si et seulement si $\Gamma \vdash P \Rightarrow Q$: autrement dit, on peut démontrer $P \Rightarrow Q$ si et seulement si on peut démontrer Q en ajoutant P aux hypothèses.

(2) Le théorème de Gödel affirme que (sous l'hypothèse $\mathsf{Consis}(\mathsf{PA})$, qui est bien affirmable dans le cadre ZFC où on travaille), $\mathsf{Consis}(\mathsf{PA})$ n'est pas démontrable dans PA ; comme on est en logique classique (arithmétique de Peano !), c'est pareil que de dire que $\neg\neg\mathsf{Consis}(\mathsf{PA})$, ou, si on préfère, $(\neg\mathsf{Consis}(\mathsf{PA})) \Rightarrow \bot$ n'est pas démontrable dans PA. D'après la question précédente, cela signifie exactement que $\bot$ n'est pas démontrable dans $\mathsf{PA} \cup \{\neg\mathsf{Consis}(\mathsf{PA})\} =: \mathsf{PA}^*$. Autrement dit, on a $\mathsf{Consis}(\mathsf{PA}^*)$: la théorie PA^* ne démontre pas de contradiction.

(3) L'affirmation $\mathsf{Consis}(\mathsf{PA}^*)$ dit qu'on ne peut pas arriver à une contradiction à partir des axiomes de Peano auxquels on a ajouté l'axiome $\neg\mathsf{Consis}(\mathsf{PA})$; en particulier, on ne peut pas arriver à une contradiction à partir des axiomes de Peano seuls, c'est-à-dire $\mathsf{Consis}(\mathsf{PA})$. On a donc $\mathsf{Consis}(\mathsf{PA}^*) \Rightarrow \mathsf{Consis}(\mathsf{PA})$ (ce fait étant démontré à partir d'arithmétique élémentaire, donc dans PA).

Comme PA^* a $\neg\mathsf{Consis}(\mathsf{PA})$ dans ses axiomes, et comme on vient d'expliquer que $\mathsf{Consis}(\mathsf{PA}^*) \Rightarrow \mathsf{Consis}(\mathsf{PA})$ (se démontre dans PA), on en déduit que $\neg\mathsf{Consis}(\mathsf{PA}^*)$ se démontre dans PA^*.

(4) La théorie PA^* *« pense »* (ou plus exactement, postule) que l'arithmétique de Peano est contradictoire, et en particulier (question (3)) qu'elle-même est contradictoire. Mais elle se trompe : ni PA ni PA^* elle-même ne sont contradictoires. Ce n'est pas absurde : c'est juste que PA^* a un axiome faux, mais le phénomène d'incomplétude énoncé par le théorème de Gödel empêche de voir que cet axiome est faux, donc d'aboutir effectivement à une contradiction. ✓

Problème 7.

Motivations : On s'intéresse dans cet exercice à la notion de *réel calculable*, qu'on va définir : c'est une formalisation possible d'un nombre réel exact pouvant être manipulé par un ordinateur. L'idée la plus évidente pour définir les réels calculables est sans doute de considérer ceux dont une écriture binaire est une fonction calculable (et de les représenter par cette fonction) : on va voir plus loin que cette définition « naïve » souffre de graves problèmes. À la place, on va définir un réel calculable comme un réel dont on peut calculer par un programme une approximation rationnelle à ε près pour n'importe quel $\varepsilon > 0$ donné en entrée (et représenter le réel par une fonction qui prend ε et renvoie l'approximation). Pour rendre cette définition plus commode à manier, on va se limiter à ε de la forme $\frac{1}{2^n}$ et renvoyer une approximation elle-même de la forme $\frac{k}{2^n}$ (cela ne change rien d'essentiel).

On introduit donc les définitions suivantes :

— Un **dyadique** est un rationnel de la forme $\frac{k}{2^n}$ avec $k \in \mathbb{Z}$ et $n \in \mathbb{N}$ (i.e., son dénominateur est une puissance de 2). Pour être plus précis, on peut appeler **dyadique de niveau** n un rationnel de la forme $\frac{k}{2^n}$ avec $k \in \mathbb{Z}$ (pour ce n-là). Noter qu'on ne demande pas que $\frac{k}{2^n}$ soit irréductible, par exemple $42 = \frac{84}{2} = \frac{168}{2^2} = \cdots$ peut être vu comme un dyadique de n'importe quel niveau.

— Si $x \in \mathbb{R}$, un **numérateur d'approximation dyadique de niveau** n de x est un entier k tel que $|k - 2^n\, x| \leq 1$. L'approximation dyadique elle-même est alors par définition $\frac{k}{2^n}$, et vérifie donc $|\frac{k}{2^n} - x| \leq \frac{1}{2^n}$.

— Un réel $x \in \mathbb{R}$ est dit **calculable** lorsqu'il y a une fonction *calculable* (totale) $\mathbb{N} \to \mathbb{Z}, n \mapsto k_n$ telle que k_n soit un numérateur d'approximation dyadique de niveau n de x. Autrement dit, il existe un programme qui, prenant n en entrée, renvoie une approximation dyadique $\frac{k_n}{2^n}$ de niveau n de x (représentée par le seul numérateur, puisque le dénominateur est par définition 2^n). Une telle fonction $n \mapsto k_n$, ou un programme qui la calcule, est dit **représenter** le réel x.

À titre d'exemple, l'entier 42 est un réel calculable puisque la fonction $n \mapsto 42 \times 2^n$ est évidemment calculable.

On prendra garde aux faits suivants :

— Un même réel x a plusieurs approximations dyadiques de niveau n (il en a même toujours 2 ou 3, puisque leurs numérateurs sont les entiers contenus dans l'intervalle $[2^n\, x - 1\,;\, 2^n\, x + 1]$ de longueur 2). À titre d'exemple, le réel 0 peut être représenté par n'importe quelle fonction calculable $n \mapsto k_n$ renvoyant un k_n dans $\{-1, 0, 1\}$.

— Même une fois fixée la fonction mathématique $n \mapsto k_n$ représentant un réel calculable x, il y a (toujours) plusieurs programmes (= fonctions informatiques) qui la calculent (« intentions »). C'est par cette dernière donnée qu'on va représenter le réel x.

On ne demande pas de justifier le fait évident qu'un programme informatique peut manipuler des données dans $\mathbb{Z}$ (entiers relatifs arbitraires), notamment faire de l'arithmétique basique dessus (sommes, différences, produits, exponentiation, division euclidienne, etc.).

Lorsqu'il est demandé d'écrire un programme, on l'écrira dans un langage de programmation raisonnable quelconque (capable de manipuler des entiers arbitrairement grands), ou sous forme de pseudocode. Par exemple, s'il est demandé d'écrire un programme représentant l'entier 42 on pourra écrire « `lambda n: 42 * 2**n` » ou « `fun n -> 42 * (pow 2 n)` » ou n'importe quoi d'aisément compréhensible de la sorte, mais en précisant ce qui n'est pas évident (par exemple que « `pow` » est une fonction calculant l'exponentiation entière). On pourra aussi librement décider si ce langage manipule des fonctions calculables sous forme de codes de Gödel de programmes les calculant, ou sous forme de fonctions informatiques (langage fonctionnel), ou même mélanger les deux au besoin (à condition de le préciser).

*

(1) Expliquer pourquoi, donnés $p \in \mathbb{Z}$ et $q \in \mathbb{N} \setminus \{0\}$ et $n \in \mathbb{N}$, on peut calculer algorithmiquement un numérateur d'approximation dyadique de niveau n de $\frac{p}{q}$. En déduire que tout rationnel est un réel calculable, et, plus précisément, écrire un programme qui, donnés p et q, renvoie une fonction représentant le rationnel $\frac{p}{q}$ comme réel calculable.

Corrigé. Si $\lfloor z \rfloor$ désigne la partie entière de z, la fonction $(p, q, n) \mapsto \lfloor 2^n\, \frac{p}{q} \rfloor$ est calculable car c'est le quotient de la division euclidienne de $2^n\, p$ par q, et il s'agit d'un numérateur d'approximation dyadique de niveau n de $\frac{p}{q}$ puisque $|\lfloor z \rfloor - z| \leq 1$. Donc, la fonction qui à p et q associe le code d'une fonction calculant $n \mapsto \lfloor 2^n\, \frac{p}{q} \rfloor$ est calculable (par le théorème s-m-n si on veut être très précis). Un

tel programme s'écrit par exemple « `fun p -> fun q -> fun n -> (p * (pow 2 n)) / q` » en supposant que « `pow` » représente l'exponentiation entière et '`/`' la division euclidienne. ✓

(2) Montrer que si x est un réel calculable alors $-x$ est calculable, et, plus précisément, écrire un programme qui, donnée une fonction qui représente x, en renvoie une qui représente $-x$. Faire de même pour $|x|$. (On rappelle à toutes fins utiles que $\Big||u|-|v|\Big| \leq |u-v|$.)

Corrigé. Si k est un numérateur d'approximation dyadique de niveau n de $x \in \mathbb{R}$, c'est-à-dire $|k-2^n\,x| \leq 1$, alors $|(-k)-2^n\,(-x)| \leq 1$ (car $|-z|=|z|$). Ceci montre que si $n \mapsto k_n$ représente x alors $n \mapsto -k_n$ représente $-x$, et elle est certainement calculable si la première l'est. Un programme calculant une fonction représentant $-x$ à partir d'une fonction représentant x s'écrit par exemple « `fun xf -> fun n -> -(xf n)` ».

Pour la valeur absolue, on procède de même en remarquant que si $|k-2^n\,x| \leq 1$, alors $\Big||k|-2^n\,|x|\Big| \leq 1$ (d'après l'inégalité qui a été rappelée). Ceci montre que si $n \mapsto k_n$ représente x alors $n \mapsto |k_n|$ représente $|x|$: un programme correspondant s'écrit par exemple « `fun xf -> fun n -> abs(xf n)` ». ✓

(3) Expliquer pourquoi, si k est un numérateur d'approximation dyadique de niveau $n+r$ de $x \in \mathbb{R}$ (avec $n, r \in \mathbb{N}$), alors k est aussi un numérateur d'approximation dyadique de niveau n de $2^r x$. En déduire que si x est un réel calculable et $r \in \mathbb{N}$ alors $2^r x$ est calculable, et, plus précisément, écrire un programme qui, donnés r et une fonction qui représente x, renvoie une fonction qui représente $2^r x$.

Corrigé. Dire que k est un numérateur d'approximation dyadique de niveau $n+r$ de x signifie par définition $|k-2^{n+r}\,x| \leq 1$. C'est exactement dire que $|k-2^n\,(2^r\,x)| \leq 1$, c'est-à-dire que k est aussi un numérateur d'approximation dyadique de niveau n de $2^r x$. On en déduit que si $n \mapsto k_n$ représente x alors $n \mapsto k_{n+r}$ représente $2^r x$. Un programme calculant une fonction représentant $2^r x$ à partir de r et d'une fonction représentant x s'écrit par exemple « `fun r -> fun xf -> fun n -> xf (n+r)` ». ✓

(4) Expliquer comment, donné $\ell \in \mathbb{Z}$, on peut trouver algorithmiquement un entier $k \in \mathbb{Z}$ tel que $[\frac{\ell}{4}-\frac{1}{2}\,;\,\frac{\ell}{4}+\frac{1}{2}] \subseteq [k-1\,;\,k+1]$ (on pourra faire un dessin de ces intervalles). En tirer une façon de trouver algorithmiquement un numérateur d'approximation dyadique de niveau n de $x+y$ à partir de numérateurs d'approximations dyadiques de niveau $n+2$ de x et y respectivement.

En déduire que si x et y sont calculables alors $x+y$ est calculable, et, plus précisément, écrire un programme qui, données des fonctions qui représentent x et y, en renvoie une qui représente $x+y$.

Corrigé. Dire $[\frac{\ell}{4}-\frac{1}{2}\,;\,\frac{\ell}{4}+\frac{1}{2}] \subseteq [k-1\,;\,k+1]$ équivaut à $[\ell-2\,;\,\ell+2] \subseteq [4k-4\,;\,4k+4]$, c'est-à-dire $4k-4 \leq \ell-2$ et $\ell+2 \leq 4k+4$, soit encore $\ell-2 \leq 4k \leq \ell+2$, autrement dit, on cherche un multiple de 4 compris au sens large entre les entiers $\ell-2$ et $\ell+2$, ce qui existe certainement car il y a 5 entiers consécutifs dans cet intervalle : pour faire ça algorithmiquement, on peut appeler $k=\lfloor\frac{\ell+2}{4}\rfloor$ le quotient de la division euclidienne de $\ell+2$ par 4, car on aura alors $4k+r=\ell+2$ avec reste $0 \leq r < 4$, donc $\ell-2 < 4k \leq \ell+2$.

Si i, j désignent des numérateurs d'approximations dyadiques de niveau $n+2$ de x et y respectivement, c'est-à-dire $|i-2^{n+2}\,x| \leq 1$ et $|j-2^{n+2}\,y| \leq 1$, on a alors $|(i+j)-2^{n+2}\,(x+y)| \leq 2$ par inégalité triangulaire, c'est-à-dire, en posant $\ell := i+j$, que $|\frac{\ell}{4}-2^n\,(x+y)| \leq \frac{1}{2}$. Or on vient de voir comment en tirer algorithmiquement un k tel que $[\frac{\ell}{4}-\frac{1}{2}\,;\,\frac{\ell}{4}+\frac{1}{2}] \subseteq [k-1\,;\,k+1]$, c'est-à-dire exactement que $|\frac{\ell}{4}-2^n\,(x+y)| \leq \frac{1}{2}$ implique $|k-2^n\,(x+y)| \leq 1$, c'est-à-dire que k est un numérateur d'approximation dyadique de niveau n de $x+y$.

On en déduit que si $n \mapsto i_n$ représente x et que $n \mapsto j_n$ représente y alors $n \mapsto \lfloor(i_{n+2}+j_{n+2}+2)/4\rfloor$ représente $x+y$. Un programme calculant une fonction représentant $x+y$ à partir de fonctions

représentant x et y s'écrit par exemple « `fun xf -> fun yf -> fun n -> ((xf (n+2)) + (yf (n+2)) + 2)/4` » (toujours en notant '`/`' la division euclidienne). ✓

(5) Montrer qu'un réel calculable z, représenté par une fonction $n \mapsto k_n$, vérifie $z \leq 0$ si et seulement on a $k_n \leq 1$ pour tout n. En déduire comment, donnée une fonction $n \mapsto k_n$ représentant z calculable, et en *supposant* $z > 0$, on peut calculer algorithmiquement un $n \in \mathbb{N}$ tel que $z \geq \frac{1}{2^n}$.

De façon analogue, montrer que, donnée une fonction $n \mapsto k_n$ représentant z calculable, et en supposant seulement $z \neq 0$, on peut décider algorithmiquement si $z < 0$ ou $z > 0$.

Corrigé. Si $k_n \leq 1$ pour tout n, alors $z \leq \frac{k_n+1}{2^n} \leq \frac{1}{2^{n-1}}$ pour tout n, donc $z \leq 0$. Réciproquement, si $z \leq 0$, alors $0 \geq z \geq \frac{k_n-1}{2^n}$ donc $k_n \leq 1$ pour tout n.

On en déduit que si on *suppose* $z > 0$, il existe un indice n tel que $k_n \geq 2$ dans toute fonction $n \mapsto k_n$ représentant z ; et pour un tel n on a alors $z \geq \frac{k_n-1}{2^n} \geq \frac{1}{2^n}$ comme recherché. Pour trouver ce n, il suffit d'effectuer une boucle infinie calculant k_n en s'arrêtant dès que $k_n \geq 2$: d'après ce qui vient d'être dit, l'hypothèse $z > 0$ garantit la terminaison de la boucle.

Si maintenant on suppose simplement $z \neq 0$, il existe soit un indice n tel que $k_n \geq 2$ soit un indice tel que $k_n \leq -2$ (par symétrie) : pour décider le signe de z, il suffit d'effectuer une boucle infinie jusqu'à ce qu'une de ces conditions soit satisfaite, et de renvoier « positif » ou « négatif » selon qu'on a trouvé un n vérifiant l'un ou l'autre. ✓

(6) (Cette question est indépendante de toutes les questions qui précèdent ou qui suivent.) Montrer qu'il n'existe pas de programme qui, donné le code d'un autre programme e, décide si ce dernier représente un réel calculable. Autrement dit, la fonction qui à e associe 1 si e calcule une fonction $n \mapsto k_n$ représentant un certain réel x, et 0 sinon, n'est pas calculable.

Corrigé. C'est une conséquence du théorème de Rice : le sous-ensemble des fonctions partielles $\mathbb{N} \dashrightarrow \mathbb{Z}$ qui sont une fonction totale représentant un réel n'est ni vide ni plein (il n'est pas vide car la fonction constante 0 est dedans, et pas plein parce que la fonction définie nulle part n'est pas dedans), donc le théorème de Rice affirme que l'ensemble des e tels que φ_e soit dans cet ensemble n'est pas décidable. ✓

(7) Soit e un programme (vu comme l'indice d'une fonction générale récursive φ_e). Soit $\eta(e)$ le réel défini de la manière suivante :

$$\eta(e) = \begin{cases} \frac{1}{2^m} & \text{si le calcul de } \varphi_e(0) \text{ termine en exactement } m \text{ étapes} \\ 0 & \text{si } \varphi_e(0)\uparrow \end{cases}$$

(Plus exactement, il faudrait plutôt écrire « $\eta(e) = \frac{1}{2^m}$ si m est le code d'un arbre de calcul pour $\varphi_e(0)$ » à la première ligne, mais on peut se permettre de confondre ces deux notions.)

Expliquer comment, donnés e et $n \in \mathbb{N}$, calculer algorithmiquement un numérateur d'approximation dyadique de niveau n de $\eta(e)$. En déduire qu'on peut écrire un programme qui, donné e, renvoie une fonction représentant le réel $\eta(e)$ (en tant réel calculable).

Corrigé. On a vu en cours qu'il est possible de tester algorithmiquement si le calcul de $\varphi_e(0)$ termine en m étapes (ou, si on préfère, si m est un arbre de calcul pour $\varphi_e(0)$). Pour calculer un numérateur d'approximation dyadique de niveau n de $\eta(e)$, on va faire ce test : on lance l'exécution de $\varphi_e(0)$ sur n étapes : si le calcul termine au bout de en $m \leq n$ étapes (ou, si on préfère, si on trouve un arbre de calcul $m \leq n$), on renvoie l'approximant $\frac{1}{2^m}$, c'est-à-dire qu'on renvoie le numérateur 2^{n-m}, qui convient car $\eta(e) = \frac{1}{2^m}$ exactement ; s'il ne termine pas dans le temps imparti, on renvoie le numérateur 0, qui convient car $\eta(e) \leq \frac{1}{2^n}$. ✓

(8) Expliquer l'erreur dans la réponse suivante à la question (7) : « On lance le calcul de $\varphi_e(0)$: s'il termine en (exactement) m étapes, on renvoie (une fonction représentant) $\frac{1}{2^m}$, qui est calculable d'après la question (1), sinon on renvoie la fonction constamment nulle. » On expliquera aussi en quoi on n'a pas commis cette erreur en répondant à ladite question.

Corrigé. L'erreur est dans le mot « sinon » : si on lance le calcul de $\varphi_e(0)$, on ne peut pas savoir à l'avance s'il va terminer : on peut certes renvoyer $\frac{1}{2^m}$ s'il termine en exactement m étapes, mais si on fait ça il n'y a pas de « sinon » possible, car on est parti dans une boule infinie. Et il n'y a pas de moyen de savoir *a priori* si la boucle terminera, car le problème de l'arrêt est indécidable (même pour $\varphi_e(0)$).

On n'a pas commis cette erreur, car on a pris *d'abord* n, qui agit comme borne sur le nombre d'étapes d'exécution : on renvoie une approximation dyadique de niveau n de $\eta(e)$, qui peut être 0 si le calcul de $\varphi_e(0)$ n'a pas terminé dans le temps imparti. ✓

(9) Déduire de la question (8) qu'il n'y a pas d'algorithme qui, donnée une fonction représentant un réel calculable x, teste si $x \neq 0$ ou $x = 0$ (i.e., renvoie « vrai » si $x \neq 0$ et « faux » si $x = 0$).

Corrigé. Si un tel algorithme existait, appliqué à $\eta(e)$ (ou plus exactement, à une fonction représentant $\eta(e)$, dont on a vu à la question (8) qu'elle se déduit algorithmiquement de e), il permettrait de tester si $\varphi_e(0)\downarrow$. Or on a vu que ceci n'est pas faisable algorithmiquement. C'est donc que l'algorithme évoqué n'existe pas. ✓

Pour la question suivante, on *admettra* le résultat suivant (plus ou moins vu en TD) : il n'existe pas d'algorithme qui, donné le code d'un programme e, termine toujours en temps fini et renvoie « vrai » si $\varphi_e(0)\downarrow = 0$ et « faux » si $\varphi_e(0)\downarrow \neq 0$ (si $\varphi_e(0)\uparrow$ il aurait le droit de répondre n'importe quoi mais toujours avec l'obligation de terminer).

(10) En modifiant un peu la construction proposée en (7) (on définira $\eta'(e)$ valant $\frac{1}{2^m}$ ou $-\frac{1}{2^m}$ ou 0 selon des cas judicieusement choisis), montrer qu'il n'est même pas possible algorithmiquement de tester si un réel calculable est ≥ 0 ou ≤ 0, i.e., il n'existe pas d'algorithme qui, donnée une fonction représentant un réel calculable x, peut renvoyer « vrai » si $x \geq 0$ ou « faux » si $x \leq 0$ (les deux réponses étant acceptables si $x = 0$).

Corrigé. On définit

$$\eta'(e) = \begin{cases} \dfrac{1}{2^m} & \text{si le calcul de } \varphi_e(0) \text{ termine en exactement } m \text{ étapes et que } \varphi_e(0) = 0 \\ -\dfrac{1}{2^m} & \text{si le calcul de } \varphi_e(0) \text{ termine en exactement } m \text{ étapes et que } \varphi_e(0) \neq 0 \\ 0 & \text{si } \varphi_e(0)\uparrow \end{cases}$$

Un argument analogue à celi de la la question (7) montre que $\eta'(e)$ est calculable à partir de e. Précisément : donné un niveau n, il suffit de lancer l'exécution de $\varphi_e(0)$ sur n étapes et, si elle termine en $m \leq n$ étapes, renvoyer l'approximant $\frac{1}{2^m}$ ou $-\frac{1}{2^m}$ (c'est-à-dire le numérateur 2^{n-m} ou -2^{n-m}) selon que le résultat qu'on a calculé est 0 ou $\neq 0$, et sinon 0. Comme en (9), décider si $\eta'(e) \leq 0$ ou $\eta'(e) \geq 0$ (en acceptant n'importe quelle réponse si c'est 0) reviendrait à décider si $\varphi_e(0)\downarrow = 0$ ou $\varphi_e(0)\downarrow \neq 0$ (en acceptant n'importe quelle réponse si $\varphi_e(0)\uparrow$, mais toujours avec l'obligation de terminer), et on a indiqué que ce n'était pas possible. ✓

Si x est un réel entre 0 et 1 (pour s'éviter le tracas de l'écriture des nombres négatifs), on rappelle qu'une **écriture binaire** de x est une suite $(u_n)_{n\geq 1}$ à valeurs dans $\{0, 1\}$ telle que $x = \sum_{n=1}^{+\infty} u_n \cdot 2^{-n}$.

Une telle suite existe toujours (par exemple, on peut prendre celle donnée $u_n = \lfloor 2^n x \rfloor - 2\lfloor 2^{n-1} x \rfloor$ où $\lfloor — \rfloor$ désigne la partie entière), mais elle n'est pas unique en général (par exemple $\frac{1}{2}$ admet les deux écritures binaires $0.100000\ldots$ et $0.011111\ldots$; en fait, ce sont exactement les dyadiques qui admettent plus d'une écriture binaire, et ils en admettent exactement deux).

On dira qu'un réel entre 0 et 1 est **naïvement calculable** lorsqu'il admet une écriture binaire $n \mapsto u_n$ calculable. On pourra aussi dire que cette fonction, ou un programme qui la calcule, le **représente naïvement**. Le but des questions suivantes est de comprendre pourquoi cette notion ne se comporte pas bien (et donc pourquoi on ne l'a pas prise comme définition principale).

(11) Montrer qu'un réel (entre 0 et 1) naïvement calculable est calculable, et plus précisément qu'il existe un algorithme qui, donnée une fonction $n \mapsto u_n$ qui calcule l'écriture binaire de x, en renvoie une représentation $n \mapsto k_n$ comme réel calculable (ainsi que définie au début de ce problème).

Corrigé. Si $x = \sum_{i=1}^{+\infty} u_i \cdot 2^{-i}$, alors $2^n\, x = \sum_{i=1}^{+\infty} u_i \cdot 2^{n-i}$ donc en appelant $k_n = \sum_{i=1}^{n} u_i \cdot 2^{n-i}$ (il s'agit d'un entier), on voit que $2^n\, x - k_n = \sum_{i=1}^{+\infty} u_{n+i} \cdot 2^{-i}$ est compris entre 0 et 1, et notamment k_n est un numérateur d'approximation dyadique de niveau n de x. Ceci fournit un algorithme calculant $n \mapsto k_n$ à partir de $n \mapsto u_n$. ✓

(12) Dans l'autre sens, en utilisant la question (10), montrer qu'il n'existe pas d'algorithme qui, donnée une fonction $n \mapsto k_n$ représentant un réel x calculable entre 0 et 1, renvoie une fonction calculable $n \mapsto u_n$ qui soit une écriture binaire de x. (*Indication :* considérer $\frac{1}{2}(\eta'(e) + 1)$.)

Corrigé. Si un tel algorithme existait, on pourrait l'appliquer à $x_e = \frac{1}{2}(\eta'(e) + 1)$, où $\eta'(e)$ est le réel calculable en fonction de e défini à la question (10) dont on ne peut pas algorithmiquement en fonction de e décider s'il est ≤ 0 ou ≥ 0. Or le premier chiffre de l'écriture binaire de x_e va être 0 ou 1 selon que $x_e \leq \frac{1}{2}$ ou $x_e \geq \frac{1}{2}$ (avec les deux possibilités si $x_e = \frac{1}{2}$), c'est-à-dire selon que $\eta'(e) \leq 0$ ou $\eta'(e) \geq 0$: on ne peut donc pas algorithmiquement calculer ce chiffre en fonction de e. ✓

(13) En utilisant de nouveau la question (10), montrer qu'il n'existe pas d'algorithme qui, donnés deux réels calculables naïfs x, y (avec, disons, x entre $\frac{1}{2}$ et 1 et y entre 0 et $\frac{1}{2}$) sous forme d'une fonction calculant une écriture binaire, renvoie $x - y$ sous cette même forme.

Corrigé. Considérons $\eta'(e)$ le réel calculable en fonction de e défini à la question (10). On peut l'écrire sous la forme $\eta'_+(e) - \eta'_-(e)$ avec $\eta'_+(e) = \frac{1}{2}(\eta'(e) + |\eta'(e)|)$ et $\eta'_-(e) = \frac{1}{2}(-\eta'(e) + |\eta'(e)|)$, tous deux positifs et tous deux valant soit 0 soit $\frac{1}{2^m}$ pour un certain m. Les questions (2) et (4) montrent qu'on peut calculer $\eta'_+(e)$ et $\eta'_-(e)$, au sens des réels calculables, en fonction de e. De plus, sachant que chacun vaut soit 0 soit $\frac{1}{2^m}$ pour un certain m (ou bien en regardant directement la construction de $\eta'(e)$), il est aussi facile d'en produire une écriture binaire : pour calculer le n-ième chiffre de son écriture binaire, si tous les précédents étaient 0, on demande une approximation dyadique de niveau $n+2$ et on renvoie 0 si elle est dans $\{-1, 0, 1\}$, sinon on renvoie 1 et ensuite uniquement des 0. Les réels $x_e = \frac{1}{2}(\eta'_+(e) + 1)$ et $y_e = \frac{1}{2}\eta'_-(e)$ sont donc tels qu'on peut calculer une écriture binaire en fonction de e, pourtant on ne peut pas en calculer pour $x_e - y_e$ comme on l'a expliqué à la question précédente. ✓

(14) Montrer que, la question (12) nonobstant, tout réel calculable (entre 0 et 1) est naïvement calculable. (On pourra distinguer deux cas : soit le réel est dyadique, soit il ne l'est pas.) On expliquera pourquoi ce n'est pas une contradiction qu'on ait montré que les réels calculables et naïvement calculables coïncident, mais qu'on peut algorithmiquement soustraire les réels calculables mais pas les réels naïvement calculables.

Corrigé. Tout réel dyadique est évidemment calculable. Il reste à montrer que si x est un réel entre 0 et 1 calculable et qui n'est pas dyadique, alors il est naïvement calculable. Pour cela, on applique

l'algorithme suivant : pour calculer le n-ième chiffre u_n de l'écriture binaire de x, on calcule une fonction représentant $2^n x - 1$, lequel est $\neq 0$ d'après l'hypothèse que x n'est pas dyadique, donc d'après la question (5) on peut décider s'il est < 0 ou > 0, et u_n vaut 0 ou 1 dans ces deux cas respectifs. Tout ceci est calculable.

Les réels calculables et naïvement calculables sont donc les mêmes en tant qu'ensembles, mais leur représentation informatiques diffèrent : on peut passer algorithmiquement de la représentation naïve par l'écriture binaire à la représentation par approximations, mais en général pas dans l'autre sens (la difficulté étant que tant qu'on est très proche d'un dyadique on ne sait jamais si on doit émettre le chiffre 0 ou 1 dans l'écriture binaire). ✓