Logique et Fondements de l'Informatique
Exercices

David A. Madore

23 janvier 2024

INF1110

Git : `29389e0 Tue Jan 23 23:02:55 2024 +0100`
(Recopier la ligne ci-dessus dans tout commentaire sur ce document)

1 Calculabilité

Exercice 1.1. ($\star\star$)
On considère la fonction $f\colon \mathbb{N} \to \mathbb{N}$ qui à $n \in \mathbb{N}$ associe le n-ième chiffre de l'écriture décimale de $\sqrt{2} \approx 1.41421356237309504880\ldots$, c'est-à-dire $f(0) = 1$, $f(1) = 4$, $f(2) = 1$, $f(3) = 4$, etc.
La fonction f est-elle calculable ? Est-elle primitive récursive ? On expliquera précisément pourquoi.

Exercice 1.2. ($\star$)
Supposons que $A \subseteq B \subseteq \mathbb{N}$. **(1)** Si B est décidable, peut-on conclure que A est décidable ? **(2)** Si A est décidable, peut-on conclure que B est décidable ?

Exercice 1.3. ($\star\star$)
(1) Soit $f\colon \mathbb{N} \to \mathbb{N}$ totale calculable. Montrer que l'image $f(\mathbb{N})$ (c'est-à-dire $\{f(i) : i \in \mathbb{N}\}$) est semi-décidable.
(2) Soit $f\colon \mathbb{N} \to \mathbb{N}$ totale calculable et strictement croissante. Montrer que l'image $f(\mathbb{N})$ (c'est-à-dire $\{f(i) : i \in \mathbb{N}\}$) est décidable.

Exercice 1.4. ($\star$)
Montrer que l'ensemble des $e \in \mathbb{N}$ tels que $\varphi_e^{(1)}(0) = \varphi_e^{(1)}(1)$ (rappel : ceci signifie que *soit* $\varphi_e^{(1)}(0) \downarrow$ et $\varphi_e^{(1)}(1) \downarrow$ et $\varphi_e^{(1)}(0) = \varphi_e^{(1)}(1)$, *soit* $\varphi_e^{(1)}(0) \uparrow$ et $\varphi_e^{(1)}(1) \uparrow$) n'est pas décidable.

Exercice 1.5. ($\star\star\star$)
(1) Soit $B \subseteq \mathbb{N}$ semi-décidable et non-vide. Montrer qu'il existe $f\colon \mathbb{N} \to \mathbb{N}$ totale calculable telle que $f(\mathbb{N}) = B$.

(*Indication :* si $m_0 \in B$ et si B est semi-décidé par le e-ième programme, i.e., $B = \{m : \varphi_e(m) \downarrow\}$, on définira $\tilde{f} \colon \mathbb{N}^2 \to \mathbb{N}$ par $\tilde{f}(n,m) = m$ si $T(n,e,\langle\langle m \rangle\rangle)$, où $T(n,e,v)$ est comme dans le théorème de la forme normale de Kleene [1], et $\tilde{f}(n,m) = m_0$ sinon. Alternativement, si on préfère raisonner sur les machines de Turing : si B est semi-décidé par la machine de Turing $\mathscr{M}$, on définit $\tilde{f}(n,m) = m$ si $\mathscr{M}$ termine sur l'entrée m en $\leq n$ étapes d'exécution, et $\tilde{f}(n,m) = m_0$ sinon.)

(2) Soit $f \colon \mathbb{N} \dashrightarrow \mathbb{N}$ partielle calculable. Montrer que l'image $f(\mathbb{N})$ (c'est-à-dire $\{f(i) : i \in \mathbb{N} \text{ et } f(i)\downarrow\}$) est semi-décidable. (*Indication :* chercher à formaliser l'idée de lancer les calculs des différents $f(i)$ « en parallèle ».)

Exercice 1.6. ($\star\star\star$)

Soit

$$T := \{e \in \mathbb{N} : \varphi_e^{(1)} \text{ est totale}\}$$

l'ensemble des codes des fonctions générales récursives totales (c'est-à-dire telles que $\forall n \in \mathbb{N}.\, (\varphi_e^{(1)}(n) \downarrow)$). On se propose de montrer que ni T ni son complémentaire $\complement T$ ne sont semi-décidables.

(1) Montrer en guise d'échauffement que T n'est pas décidable.

(2) Soit $H := \{d \in \mathbb{N} : \varphi_d^{(1)}(0) \downarrow\}$ (variante du problème de l'arrêt). Rappeler brièvement pourquoi H est semi-décidable mais non décidable, et pourquoi son complémentaire $\complement H$ n'est pas semi-décidable.

(3) Montrer qu'il existe une fonction $\rho \colon \mathbb{N} \to \mathbb{N}$ (totale) calculable (d'ailleurs même p.r.) telle que $\varphi_d^{(1)}(0) \downarrow$ si et seulement si $\varphi_{\rho(d)}^{(1)}$ est totale (*indication :* on pourra par exemple construire un programme e qui ignore son argument et qui simule d sur l'entrée 0). Reformuler cette affirmation comme une réduction. En déduire que le complémentaire $\complement T$ de T n'est pas semi-décidable.

(4) Montrer qu'il existe une fonction $\sigma \colon \mathbb{N} \to \mathbb{N}$ (totale) calculable (d'ailleurs même p.r.) telle que $\varphi_d^{(1)}(0) \downarrow$ si et seulement si $\varphi_{\sigma(d)}^{(1)}$ *n'est pas* totale (*indication :* on pourra par exemple construire un programme e qui lance d sur l'entrée 0 pour un nombre d'étapes donné en argument, et fait une boucle infinie si cette exécution termine avant le temps imparti). Reformuler cette affirmation comme une réduction. En déduire que T n'est pas semi-décidable.

Exercice 1.7. ($\star\star\star$)

Soit $f \colon \mathbb{N} \to \mathbb{N}$ une fonction totale : montrer qu'il y a équivalence entre les affirmations suivantes :

1. la fonction f est calculable,
2. le graphe $\Gamma_f := \{(i, f(i)) : i \in \mathbb{N}\} = \{(i,q) \in \mathbb{N}^2 : q = f(i)\}$ de f est décidable,
3. le graphe Γ_f de f est semi-décidable.

(Montrer que (3) implique (1) est le plus difficile : on pourra commencer par s'entraîner en montrant que (2) implique (1). Pour montrer que (3) implique (1), on pourra chercher une façon de tester en parallèle un nombre croissant de valeurs de q de manière à s'arrêter si l'une quelconque convient. On peut s'inspirer de l'exercice 1.5 question (2).)

Exercice 1.8. ($\star\star\star$)

Si $e \mapsto \psi_e^{(1)}$ est la numérotation standard des fonctions primitives récursives en une variable (= d'arité 1) et $e \mapsto \varphi_e^{(1)}$ celle des fonctions générales récursives en une variable, on considère les ensembles

$$M := \{e \in \mathbb{N} : \psi_e^{(1)} \text{ définie}\}$$

1. Rappel : c'est-à-dire que $T(n,e,\langle\langle \underline{x} \rangle\rangle)$ signifie : « n est le code d'un arbre de calcul de $\varphi_e(\underline{x})$ termine » (le résultat $\varphi_e(\underline{x})$ du calcul étant alors noté $U(n)$).

$$N := \{e \in \mathbb{N} : \exists e' \in \mathbb{N}.(\psi^{(1)}_{e'} \text{ définie et } \varphi^{(1)}_e = \psi^{(1)}_{e'})\}$$

Expliquer informellement ce que signifient ces deux ensembles (en insistant sur le rapport entre eux), dire s'il y a une inclusion de l'un dans l'autre, et dire si l'un ou l'autre est décidable.

Exercice 1.9. ($\star\star\star\star$)

On considère la fonction $f\colon \mathbb{N}^2 \to \mathbb{N}$ qui à (e, x) associe 1 si $\psi^{(1)}_e(x) = 0$, et 0 sinon (y compris si $\psi^{(1)}_e$ n'est pas définie) ; ici, $e \mapsto \psi^{(1)}_e$ est la numérotation standard des fonctions primitives récursives en une variable (= d'arité 1).

La fonction f est-elle calculable ? Est-elle primitive récursive ? On expliquera précisément pourquoi. (On s'inspirera de résultats vus en cours.) Cela changerait-il si on inversait les valeurs 0 et 1 dans f ?

Exercice 1.10. ($\star\star\star\star$)

Soit

$$Z := \{e \in \mathbb{N} : \exists n \in \mathbb{N}.\, (\psi^{(1)}_e(n) = 0)\}$$

l'ensemble des codes e des fonctions p.r. $\mathbb{N} \to \mathbb{N}$ qui prennent (au moins une fois) la valeur 0 (ici, $e \mapsto \psi^{(1)}_e$ est la numérotation standard des fonctions primitives récursives en une variable).

Montrer que Z est semi-décidable. Montrer qu'il n'est pas décidable.

Exercice 1.11. ($\star\star\star$)

On rappelle que le mot « configuration », dans le contexte de l'exécution d'une machine de Turing, désigne la donnée de l'état interne de la machine, de la position de la tête de lecture, et de la totalité de la bande. (Et la « configuration vierge » est la configuration où l'état est 1, la tête est à la position 0, et la bande est entièrement remplie de 0.)

On considère l'ensemble $\mathscr{F}$ des machines de Turing M dont l'exécution, à partir de la configuration vierge C_0, conduit à un nombre fini de configurations distinctes (i.e., si on appelle $C^{(n)}$ la configuration atteinte au bout de n étapes d'exécution en démarrant sur C_0, on demande que l'ensemble $\{C^{(n)} : n \in \mathbb{N}\}$ soit fini).

(1) Montrer que $\mathscr{F}$ est semi-décidable. (*Indication :* on pourra commencer par remarquer, en le justifiant, que « passer par un nombre fini de configurations distinctes » équivaut à « terminer ou revenir à une configuration déjà atteinte ».)

(2) Montrer que $\mathscr{F}$ n'est pas décidable. (*Indication :* si on savait décider $\mathscr{F}$ on saurait décider le problème de l'arrêt.)

Exercice 1.12. ($\star\star\star\star$)

Soit $f\colon \mathbb{N} \to \mathbb{N}$ une fonction calculable par une machine de Turing en *complexité d'espace* primitive récursive : cela signifie qu'il existe $p\colon \mathbb{N} \to \mathbb{N}$ primitive récursive et une machine de Turing $\mathscr{M}$ telle que si on lui présente $n \in \mathbb{N}$ en entrée (écrit avec les conventions usuelles, c'est-à-dire en unaire sur la bande), la machine s'arrête en temps fini en ayant écrit $f(n)$ sur la bande, et de plus le nombre de cases de la bande que la tête de lecture a parcourues est $\leq p(n)$ (c'est-à-dire que $\mathscr{M}$ a utilisé $\leq p(n)$ cellules mémoire pour faire le calcul).

On veut montrer que f elle-même est primitive récursive (c'est-à-dire qu'une fonction calculable en complexité d'espace p.r. est elle-même p.r., de la même manière qu'une fonction calculable en complexité en temps p.r. est elle-même p.r.).

(1) Si $\mathscr{M}$ a $\leq m$ états, montrer que le calcul de $f(n)$ par $\mathscr{M}$ ne peut faire intervenir qu'au plus $m \times p(n) \times 2^{p(n)+n}$ configurations différentes (on rappelle qu'une *configuration* est la donnée d'un état, d'une position de la tête, et de la valeur de chaque cellule du ruban).

(2) En déduire que le calcul de $f(n)$ par $\mathscr{M}$ termine en au plus $m \times p(n) \times 2^{p(n)+n}$ étapes (*indication :* sinon le calcul bouclerait indéfiniment, et on a supposé que ce n'était pas le cas).

(3) En déduire que f est primitive récursive.

Exercice 1.13. ($\star\star\star$)

On s'intéresse à des tableaux d'entiers naturels, indicés par les entiers naturels, dont toutes les valeurs valent 0 sauf un nombre fini (i.e., des fonctions $\tau : \mathbb{N} \to \mathbb{N}$ dont le support $\{i \in \mathbb{N} : \tau(i) \neq 0\}$ est fini). Un tel tableau τ sera Gödel-codé comme un entier naturel sous la forme (disons) de la liste $\langle\langle\langle i_1, \tau(i_1)\rangle, \ldots, \langle i_k, \tau(i_k)\rangle\rangle\rangle$ où $i_1 < i_2 < \cdots < i_k$ sont les indices tels que la valeur $\tau(i)$ dans le tableau à cet indice soit $\neq 0$.

(1) Sans rentrer dans énormément de détails, expliquer pourquoi la fonction $(\tau, i) \mapsto \tau(i)$ (« lecture du tableau τ à l'indice i ») et $(\tau, i, v) \mapsto \tau'$ où $\tau'(j) = \tau(j)$ sauf $\tau'(i) = v$ (« modification du tableau τ à l'indice i pour y mettre la valeur v ») sont primitives récursives.

(2) Sans rentrer dans énormément de détails, en déduire pourquoi, du coup, un algorithme primitif récursif peut utiliser un tableau dans une boucle où elle lit et écrit des valeurs arbitraires du tableau.

Exercice 1.14. ($\star\star\star\star\star$)

On rappelle la définition de la fonction d'Ackermann $A : \mathbb{N}^3 \to \mathbb{N}$:

$$\begin{aligned} A(m, n, 0) &= m + n \\ A(m, 0, 1) &= 0 \\ A(m, 0, k) &= 1 \text{ si } k \geq 2 \\ A(m, n+1, k+1) &= A(m,\ A(m, n, k+1),\ k) \end{aligned}$$

On a vu en cours que cette fonction est calculable mais non primitive récursive. On admettra sans discussion que $A(m, n, k)$ est croissante en chaque variable dès que $m \geq 2$ et $n \geq 2$. On pourra aussi utiliser sans discussion les faits suivants :

$$\begin{aligned} A(m, n, 1) &= mn \\ A(m, n, 2) &= m^n \\ A(0, n, k) &= ((n+1)\%2) \text{ si } k \geq 3 \\ A(1, n, k) &= 1 \text{ si } k \geq 2 \\ A(m, 1, k) &= m \text{ si } k \geq 1 \\ A(2, 2, k) &= 4 \end{aligned}$$

On considérera aussi la *fonction indicatrice du graphe* de A, c'est-à-dire la fonction $B : \mathbb{N}^4 \to \mathbb{N}$:

$$\begin{aligned} B(m, n, k, v) &= 1 \quad \text{si } v = A(m, n, k) \\ B(m, n, k, v) &= 0 \quad \text{sinon} \end{aligned}$$

(1) Écrire un algorithme qui calcule $A(m, n, k)$ à partir de m, n, k et d'un *majorant* b de $A(m, n, k)$ selon le principe suivant : si $m \geq 2$ et $n \geq 2$, pour chaque ℓ allant de 0 à k et chaque i allant de 0 à b (bien noter : c'est b et pas n ici), on calcule $A(m, i, \ell)$, et on la stocke dans la case (i, ℓ) d'un tableau, à condition que les valeurs déjà calculées et contenues dans le tableau permettent de la calculer (pour les petites valeurs $m \leq 1$ ou $n \leq 1$ on utilise les formules données ci-dessus ; sinon, on essaye d'utiliser la formule de récurrence en consultant le tableau). Expliquer pourquoi la valeur $A(m, n, k)$ est bien calculée par cet algorithme.

(2) Expliquer pourquoi l'algorithme qu'on a écrit en (1) est primitif récursif (on pourra prendre connaissance des conclusions de l'exercice 1.13).

(3) En déduire que la fonction B est primitive récursive.

(Autrement dit, on ne peut pas calculer A par un algorithme p.r., mais on peut *vérifier* sa valeur, si elle est donnée en entrée, par un tel algorithme.)

Exercice 1.15. ($\star\star$)

On dira que deux parties L, M de $\mathbb{N}$ disjointes (c'est-à-dire $L \cap M = \varnothing$) sont **calculablement séparables** lorsqu'il existe un $E \subseteq \mathbb{N}$ décidable tel que $L \subseteq E$ et $M \subseteq \complement E$ (où $\complement E$ désigne le complémentaire de E) ; dans le cas contraire, on les dit **calculablement inséparables**.

(1) Expliquer pourquoi deux ensembles $L, M \subseteq \mathbb{N}$ disjoints sont calculablement séparables si et seulement s'il existe un algorithme qui, prenant en entrée un élément x de $\mathbb{N}$:

— termine toujours en temps fini,

— répond « vrai » si $x \in L$ et « faux » si $x \in M$ (rien n'est imposé si $x \notin L \cup M$).

(2) Expliquer pourquoi deux ensembles *décidables* disjoints sont toujours calculablement séparables.

On cherche maintenant à montrer qu'il existe deux ensembles $L, M \subseteq \mathbb{N}$ *semi-décidables* disjoints et calculablement *in*séparables.

Pour cela, on appelle $L := \{\langle e, x\rangle : \varphi_e(x)\!\downarrow = 1\}$ l'ensemble des codes des couples $\langle e, x\rangle$ formés d'un programme (=algorithme) e et d'une entrée x, tels que l'exécution du programme e sur l'entrée x termine en temps fini et renvoie la valeur 1 ; et $M := \{\langle e, x\rangle : \varphi_e(x)\!\downarrow = 2\}$ l'ensemble défini de la même manière mais avec la valeur 2.

(3) Pourquoi L et M sont-ils disjoints ?

(4) Pourquoi L et M sont-ils semi-décidables ?

(5) En imitant la démonstration du théorème de Turing sur l'indécidabilité du problème de l'arrêt, ou bien en utilisant le théorème de récursion de Kleene, montrer qu'il n'existe aucun algorithme qui, prenant en entrée le code d'un couple $\langle e, x\rangle$, termine toujours en temps fini et répond « vrai » si $\langle e, x\rangle \in L$ et « faux » si $\langle e, x\rangle \in M$ (*indication :* si un tel algorithme existait, on pourrait s'en servir pour faire le contraire de ce qu'il prédit).

(6) Conclure.

Exercice 1.16. ($\star\star\star$)

Dans cet exercice, on suppose qu'on doit faire un choix entre deux options qu'on appelera « X » et « Y ». L'un de ces choix est le « bon » choix et l'autre est le « mauvais » choix, mais on ignore lequel est lequel.

On dispose d'un programme p ayant la propriété garantie suivante : si on fournit en entrée à p un programme q (ne prendant, lui, aucune entrée) qui termine et renvoie le bon choix, alors p lui aussi termine et renvoie le bon choix. (En revanche, si q fait autre chose que renvoyer le bon choix, que ce soit parce qu'il ne termine pas, parce qu'il renvoie le mauvais choix, ou qu'il renvoie autre chose que X ou Y, alors p appelé sur q peut faire n'importe quoi, y compris ne pas terminer, renvoyer le mauvais choix, ou renvoyer autre chose que X ou Y.)

(1) Expliquer comment construire un programme q tel que p appelé sur l'entrée q *ne peut pas renvoyer le mauvais choix* (il se peut qu'il ne termine pas, ou qu'il renvoie autre chose que X ou Y, mais il ne peut pas terminer et renvoyer la mauvais choix).

(*Indication :* utiliser l'astuce de Quine, c'est-à-dire le théorème de récursion de Kleene, en s'inspirant d'une démonstration de l'indécidabilité du problème de l'arrêt ou du théorème de Rice.)

(2) Expliquer pourquoi il n'y a pas moyen, avec le p qu'on nous a fourni, mais sans connaître le bon choix, de faire un programme qui termine à coup sûr et renvoie le bon choix.

(*Indication :* raisonner par symétrie en proposant un p qui marchera quel que soit le bon choix.)

Exercice 1.17. ($\star\star\star$)

(1) Montrer qu'il existe un programme p tel que $\varphi_p(q) = \varphi_q(q)$ pour tout q et que $\varphi_p(p) = 42$ (autrement dit, quand on lui fournit en entrée un autre programme q, le programme p exécute q sur q, mais par ailleurs p exécuté sur lui-même doit renvoyer 42). On expliquera soigneusement l'utilisation du théorème de récursion de Kleene ici.

(2) On construit p' de façon analogue à la question précédente, mais en remplaçant 42 par 1729. Que donne le résultat de p exécuté sur p' ? Et de p' exécuté sur p ? Que dire des fonctions partielles φ_p et $\varphi_{p'}$?

Exercice 1.18. ($\star$)

Soit $h\colon \mathbb{N} \dashrightarrow \mathbb{N}$ une fonction partielle calculable. Montrer qu'il existe une fonction primitive récursive H qui à $e \in \mathbb{N}$ associe un e' tel que $\varphi_{e'} = h \circ \varphi_e$ (c'est-à-dire que $\varphi_{e'}(n) = h(\varphi_e(n))$, à condition que $m := \varphi_e(n)$ soit défini et que $h(m)$ le soit). On rédigera très soigneusement.

2 λ-calcul non typé

Exercice 2.1. ($\star$)

Pour chacun des termes suivants du λ-calcul non typé, dire s'il est en forme normale, ou en donner la forme normale s'il y en a une. **(a)** $(\lambda x.x)(\lambda x.x)$ **(b)** $(\lambda x.xx)(\lambda x.x)$ **(c)** $(\lambda x.xx)(\lambda x.xx)$ **(d)** $(\lambda xx.x)(\lambda xx.x)$ **(e)** $(\lambda xy.x)(\lambda xy.x)$ **(f)** $(\lambda xy.xy)y$ **(g)** $(\lambda xy.xy)(\lambda xy.xy)$

Exercice 2.2. ($\star\star$)

(1) Considérons le terme $T_2 := (\lambda x.xxx)(\lambda x.xxx)$ du λ-calcul non typé. Étudier le graphe des β-réductions dessus, c'est-à-dire tous les termes obtenus par β-réduction à partir de T_2, et les β-réductions entre eux.

(2) Que se passe-t-il pour $V := (\lambda x.x(xx))(\lambda x.x(xx))$? Sans entrer dans les détails, on donnera quelques chemins de β-réduction, notamment celui suivi par la réduction extérieure gauche.

(3) Étudier de façon analogue le comportement du terme $R := (\lambda x.\lambda v.xxv)(\lambda x.\lambda v.xxv)$ sous l'effet de la β-réduction.

Exercice 2.3. ($\star\star\star$)

On considère la traduction évidente des termes du λ-calcul en langage Python et/ou en Scheme définie de la manière suivante :

- une variable se traduit en elle-même (i.e., en l'identificateur de ce nom),
- une application (PQ) du λ-calcul se traduit par `P(Q)` pour le Python et par `(P Q)` pour le Scheme (dans les deux cas, c'est la notation pour l'application d'une fonction à un terme), où `P`, `Q` sont les traductions de P, Q respectivement,
- une abstraction $\lambda v.E$ du λ-calcul se traduit par `(lambda v: E)` en Python et `(lambda (v) E)` en Scheme (dans les deux cas, c'est la notation pour la création d'une fonction anonyme), où `E` est la traduction de E et `v` l'identificateur ayant pour nom celui de la variable v.

(a) Traduire les entiers de Church $\overline{0}, \overline{1}, \overline{2}, \overline{3}$ en Python et en Scheme.

(b) Écrire une fonction dans chacun de ces langages prenant en entrée (la conversion d')un entier de Church et renvoyant l'entier natif (c'est-à-dire au sens usuel du langage) correspondant. On pourra pour cela utiliser la fonction successeur qui s'écrit `(lambda n: n+1)` en Python et `(lambda (n) (+ n 1))` en Scheme.

(c) Traduire les fonctions $\lambda mnfx.nf(mfx)$, $\lambda mnf.n(mf)$ et $\lambda mn.nm$ qui représentent $(m, n) \mapsto m + n$, $(m, n) \mapsto mn$ et $(m, n) \mapsto m^n$ sur les entiers de Church en Python et en Scheme, et vérifier leur bon fonctionnement sur quelques exemples (en utilisant la fonction écrite en (b) pour décoder le résultat).

(d) Traduire le terme non-normalisable $(\lambda x.xx)(\lambda x.xx)$ en Python et Scheme : que se passe-t-il quand on le fait exécuter à un interpréteur de ces langages ? Expliquer brièvement cette différence.

(e) Proposer une tentative de traduction des termes du λ-calcul en OCaml ou Haskell : reprendre les questions précédentes en indiquant ce qui change pour ces langages.

Exercice 2.4. ($\star\star$)

On s'intéresse à une façon d'implémenter les couples en λ-calcul non-typé : $\Pi := \lambda xyf.fxy$ (servant à faire un couple) et $\pi_1 := \lambda p.p(\lambda xy.x)$ et $\pi_2 := \lambda p.p(\lambda xy.y)$ (servant à en extraire la première et la seconde composantes).

(1) Montrer que, pour tous termes X, Y, le terme $\pi_1(\Pi XY)$ se β-réduit en X et $\pi_2(\Pi XY)$ se β-réduit en Y.

(2) Expliquer intuitivement comment fonctionnent Π, π_1, π_2 : comment est représentée le couple (x, y) par Π (c'est-à-dire Πxy) ?

(3) Écrire les fonctions Π, π_1, π_2 (on pourra les appeler par exemple `pairing`, `proj1`, `proj2`) dans un langage de programmation fonctionnel (on pourra prendre connaissance de l'énoncé de l'exercice 2.3), et vérifier leur bon fonctionnement. (Mieux vaut, ici, choisir un langage fonctionnel non typé, c'est-à-dire dynamiquement typé, pour mieux refléter le λ-calcul non typé et éviter d'éventuels tracas liés au typage. Si le langage a des couples natifs, on pourra écrire des conversions des couples natifs dans le codage défini ici, et vice versa.) Si on a des notions de compilation : sous quelle forme est stockée l'information du couple dans la représentation faite par Π ?

3 Correspondance de Curry-Howard et calcul propositionnel intuitionniste

Exercice 3.1. ($\star\star$)

Pour chacune des preuves suivantes écrites informellement en langage naturel dans le calcul propositionnel, écrire le λ-terme de preuve (c'est-à-dire le terme du λ-calcul propositionnel simplement typé étendu d'un type $\bot$ ayant pour type la proposition prouvée) qui lui correspond. Ces raisonnements sont-ils intuitionnistement valables ? Qu'en conclut-on ?

(a) On va prouver $\neg\neg(A \Rightarrow B) \Rightarrow \neg\neg A \Rightarrow \neg\neg B$. Pour cela, supposons $\neg\neg(A \Rightarrow B)$ et $\neg\neg A$ et $\neg B$ et on veut arriver à une contradiction. Supposons $A \Rightarrow B$. Alors si on a A, on a B, ce qui contredit $\neg B$; donc $\neg A$: mais ceci contredit $\neg\neg A$. Donc $\neg(A \Rightarrow B)$. Mais ceci contredit $\neg\neg(A \Rightarrow B)$, comme annoncé.

(b) On va prouver $(A \Rightarrow \neg\neg B) \Rightarrow \neg\neg(A \Rightarrow B)$. Supposons $A \Rightarrow \neg\neg B$ ainsi que $\neg(A \Rightarrow B)$ et on veut arriver à une contradiction. Si on a B, alors certainement $A \Rightarrow B$, ce qui contredit

$\neg(A \Rightarrow B)$: ceci montre $\neg B$. Si on a A, alors notre hypothèse $A \Rightarrow \neg\neg B$ nous donne $\neg\neg B$, d'où une contradiction, et notamment B. On a donc prouvé $A \Rightarrow B$, d'où la contradiction recherchée.

(c) On va prouver $(\neg\neg A \Rightarrow \neg\neg B) \Rightarrow (A \Rightarrow \neg\neg B)$. Mais on sait que A implique $\neg\neg A$, donc $\neg\neg A \Rightarrow C$ implique $A \Rightarrow C$ (quel que soit C, et notamment pour $\neg\neg B$).

Exercice 3.2. ($\star\star\star$)

En utilisant une fonction call/cc (typé comme la loi de Peirce), construire un terme de type $(A \wedge B \Rightarrow C) \Rightarrow (A \Rightarrow C) \vee (B \Rightarrow C)$ dont le comportement en tant que programme est décrit informellement comme suit : donné f de type $A \wedge B \Rightarrow C$, pour construire une valeur de type $(A \Rightarrow C) \vee (B \Rightarrow C)$, on capture une continuation (disons k) et on renvoie « provisoirement » une fonction $A \Rightarrow C$ qui attend un paramètre x de type A et qui, quand elle le reçoit, invoque la contination k pour « revenir en arrière dans le temps » renvoyer finalement une fonction $B \Rightarrow C$ qui prend en entrée y de type B et renvoie $f\langle x, y\rangle$.

Exercice 3.3. ($\star$)

Montrer que la formule (de Gödel-Dummett)

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

est prouvable en calcul propositionnel classique et ***n'est pas*** prouvable en calcul propositionnel intuitionniste.

Exercice 3.4. ($\star$)

Montrer en calcul propositionnel intuitionniste que

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

Exercice 3.5. ($\star\star$)

(1) Montrer en calcul propositionnel intuitionniste que $(A \vee \neg A) \Rightarrow (\neg\neg A \Rightarrow A)$.

(2) Montrer que $(\neg\neg A \Rightarrow A) \Rightarrow (A \vee \neg A)$ n'est pas démontrable en calcul propositionnel intuitionniste. *(Cette question est sans doute plus facile à traiter en utilisant l'une quelconque des sémantiques vues en cours pour le calcul propositionnel intuitionniste.)*

(3) Montrer qu'il revient pourtant au même, en calcul propositionnel intuitionniste, de postuler $P \vee \neg P$ pour *toute* proposition P, ou bien de postuler $\neg\neg Q \Rightarrow Q$ pour *toute* proposition Q. (Pour le sens qui ne découle pas immédiatement de (1), on pourra démontrer la proposition $\neg\neg(P \vee \neg P)$ sans hypothèse.)

Exercice 3.6. ($\star\star$)

Montrer $\neg\neg A \wedge \neg\neg B \Rightarrow \neg\neg(A \wedge B)$ en calcul propositionnel intuitionniste. On écrira la preuve très soigneusement et on en donnera un λ-terme.

Exercice 3.7. ($\star\star\star$)

Montrer qu'il revient pourtant au même, en calcul propositionnel intuitionniste, de postuler $\neg P \vee \neg\neg P$ pour *toute* proposition P (« loi faible du tiers exclu »), ou bien de postuler $\neg(Q \wedge R) \Rightarrow \neg Q \vee \neg R$ pour *toutes* propositions Q, R (« quatrième loi de De Morgan »). On pourra prendre connaissance de la conclusion de l'exercice 3.6.

Expliquer pourquoi $\neg(A \wedge B) \Rightarrow \neg A \vee \neg B$ n'est pas démontrable en calcul propositionnel intuitionniste.

Exercice 3.8. ($\star\star$)

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 :

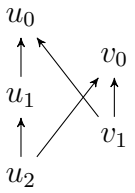
$$\lambda k.\, k\,(k\,(\lambda x.x)\,(\lambda y.y))$$

(autrement dit, `fun k -> k (k (fun x -> x) (fun y -> y))` en syntaxe OCaml).

4 Sémantique(s) du calcul propositionnel intuitionniste

Exercice 4.1. ($\star$)

On considère le cadre de Kripke dessiné ci-dessous, où une flèche $w \to w'$ signifie que $w \leq w'$ (« le monde w' est accessible depuis le monde w »), sachant que la relation $\leq$ doit bien sûr être réflexive et transitive (c'est la clôture réflexive-transitive de celle qui est représentée par les flèches : c'est-à-dire qu'on a bien sûr $u_0 \leq u_0$ et $u_2 \leq u_0$ par exemple, malgré l'absence de flèches explicites pour le rappeler).



Soit p l'affectation de vérité qui vaut 1 en u_0 et 0 en chacun de v_0, u_1, v_1, u_2. Pour ce p, calculer l'affectation de vérité de $((\neg\neg p \Rightarrow p) \Rightarrow (p \vee \neg p)) \Rightarrow (\neg p \vee \neg\neg p)$ (c'est-à-dire plus exactement $((\dot\neg\,\dot\neg\, p \mathbin{\dot\Rightarrow} p) \mathbin{\dot\Rightarrow} (p \mathbin{\dot\vee} \dot\neg\, p)) \mathbin{\dot\Rightarrow} (\dot\neg\, p \mathbin{\dot\vee} \dot\neg\,\dot\neg\, p)$, où les points rappellent qu'on parle de l'interprétation des connecteurs données par la sémantique de Kripke). En déduire que la formule $((\neg\neg A \Rightarrow A) \Rightarrow (A \vee \neg A)) \Rightarrow (\neg A \vee \neg\neg A)$ n'est pas démontrable en calcul propositionnel intuitionniste.

Exercice 4.2. ($\star\star$)

Pour $n \in \mathbb{N}$, on considère le cadre de Kripke $\{w_0, \ldots, w_{n-1}\}$ formé de n mondes totalement ordonnés par $w_i \leq w_j$ lorsque $i \geq j$ (le fait d'inverser l'ordre s'avérera plus commode pour l'écriture de la suite) :

$$w_{n-1} \longrightarrow \cdots \longrightarrow w_2 \longrightarrow w_1 \longrightarrow w_0$$

On définit aussi $n+1$ affectations de vérité $p_0, \ldots, p_n$ par $p_k(w_i) = 1$ lorsque $i < k$ et $p_k(w_i) = 0$ lorsque $i \geq k$ (notamment, p_0 est l'affectation $\dot\bot$ et p_n est l'affectation $\dot\top$). Pourquoi sont-ce les seules affectations de vérité sur ce cadre ? Calculer les tableaux de $\dot\wedge, \dot\vee, \dot\Rightarrow, \dot\neg$ sur $p_0, \ldots, p_n$.

Donner un exemple de formule propositionnelle classiquement démontrable et qui n'est pas validée par ce cadre (si $n \geq 2$), et un exemple de formule propositionnelle validée par ce cadre et qui pas démontrable intuitionnistement.

Exercice 4.3. ($\star\star\star\star$)

En prenant connaissance du résultat de l'exercice 1.15, montrer que la formule suivante (« axiome de Kreisel-Putnam ») n'est pas réalisable :

$$(\neg A \Rightarrow B \vee C) \Rightarrow (\neg A \Rightarrow B) \vee (\neg A \Rightarrow C)$$

(On pourra supposer par l'absurde qu'il y a un programme r qui réalise cette formule, et chercher à s'en servir pour séparer les ensembles L et M définis dans l'exercice 1.15. *Indication :* plus précisément, on pourra poser B_z comme valant $\mathbb{N}$ si $z \in L$ et $\varnothing$ sinon ; C_z comme valant $\mathbb{N}$ si $z \in M$ et $\varnothing$ sinon ; et A_z comme valant $\varnothing$ si $z \in L \cup M$ et $\mathbb{N}$ sinon ; et chercher à définir un élément de $(\dot\neg\, A_z \mathrel{\dot\Rightarrow} B_z \mathrel{\dot\vee} C_z)$ auquel appliquer r.)

Exercice 4.4. ($\star\star\star$)

(1) Donné un problème fini (X, S), décrire soigneusement le problème $\dot\neg(X, S)$ (combien de candidats il a, et combien de solutions : on distinguera $S = \varnothing$ ou $S \neq \varnothing$).

(2) Expliquer pourquoi la formule suivante (« axiome de Kreisel-Putnam ») est valable dans la sémantique de Medvedev des problèmes finis :

$$(\neg A \Rightarrow B \vee C) \Rightarrow (\neg A \Rightarrow B) \vee (\neg A \Rightarrow C)$$

(On décrira explicitement le problème fini décrit par la partie gauche et par la partie droite de l'implication centrale de cette formule, et leurs solutions.)

Exercice 4.5. ($\star$)

En considérant les parties suivantes de $\mathbb{R}^2$:

$$\begin{array}{ll} U_1 = \{(x,y) \in \mathbb{R}^2 : x < 0\} & U_2 = \{(x,y) \in \mathbb{R}^2 : x > 0\} \\ V_1 = \{(x,y) \in \mathbb{R}^2 : y > -x^2\} & V_2 = \{(x,y) \in \mathbb{R}^2 : y < x^2\} \end{array}$$

(faire un dessin !) montrer que la formule de Tseitin

$$\begin{array}{c} (\neg(A_1 \wedge A_2) \wedge (\neg A_1 \Rightarrow (B_1 \vee B_2)) \wedge (\neg A_2 \Rightarrow (B_1 \vee B_2))) \\ \Rightarrow ((\neg A_1 \Rightarrow B_1) \vee (\neg A_2 \Rightarrow B_1) \vee (\neg A_1 \Rightarrow B_2) \vee (\neg A_2 \Rightarrow B_2)) \end{array}$$

n'est pas démontrable en calcul propositionnel intuitionniste.

5 Quantificateurs

Exercice 5.1. ($\star\star\star$)

On appelle **système F** de Girard (dit aussi $\lambda 2$), ou plus exactement le système F étendu par types produits, sommes, 1, 0 et quantificateur existentiel, le système de logique et/ou typage de la manière décrite ci-dessous. L'idée générale est que le système F ajoute au calcul propositionnel intuitionniste la capacité à quantifier sur des propositions ; ou, si on voit ça comme un système de typage, un polymorphisme explicite.

On part des règles du λ-calcul simplement typé étendu par types produits, sommes, 1, 0, qu'on notera avec les notations logiques $(\wedge, \vee, \top, \bot)$, ou, ce qui revient au même, du calcul propositionnel intuitionniste. On ajoute un symbole spécial $*$ pour représenter le « type des propositions » (ou types) avec les règles de typage suivantes définissant la construction des propositions (ou types) :

$$\frac{\Gamma \vdash P : * \qquad \Gamma \vdash Q : *}{\Gamma \vdash P \Rightarrow Q : *} \qquad \frac{\Gamma \vdash P : * \qquad \Gamma \vdash Q : *}{\Gamma \vdash P \wedge Q : *} \qquad \frac{\Gamma \vdash P : * \qquad \Gamma \vdash Q : *}{\Gamma \vdash P \vee Q : *}$$

$$\frac{}{\Gamma \vdash \top : *} \qquad \frac{}{\Gamma \vdash \bot : *}$$

On ajoute des règles permettant de quantifier sur $*$:

$$\frac{\Gamma,\ T : * \ \vdash\ Q : *}{\Gamma\ \vdash\ (\forall(T : *).\, Q) : *} \qquad \frac{\Gamma,\ T : * \ \vdash\ Q : *}{\Gamma\ \vdash\ (\exists(T : *).\, Q) : *}$$

Et on ajoute les règles d'introduction et d'élimination de des quantificateurs :

$$\frac{\Gamma,\ T : * \ \vdash\ s : Q}{\Gamma \vdash\ (\lambda(T : *).\, s)\ :\ (\forall(T : *).\, Q)} \qquad \frac{\Gamma \vdash\ f\ :\ (\forall(T : *).\, Q) \qquad \Gamma \vdash P : *}{\Gamma \vdash\ fP\ :\ Q[T\backslash P]}$$

$$\frac{\Gamma \vdash P : * \qquad \Gamma \vdash\ z\ :\ Q[T\backslash P]}{\Gamma \vdash\ \langle P, z\rangle\ :\ (\exists(T : *).\, Q)} \qquad \frac{\Gamma \vdash\ z\ :\ (\exists(T : *).\, P) \qquad \Gamma,\ T : *,\ h : P\ \vdash\ Q}{\Gamma \vdash (\texttt{match}\ \ z\ \ \texttt{with}\ \ \langle T, h\rangle \mapsto s)\ :\ Q}$$

Pour abréger les notations, on écrit souvent, et on le fera dans la suite, « ΛT » plutôt que « $\lambda(T : *)$ » (abstraction sur les propositions/types), et « $\forall T$ » / « $\exists T$ » plutôt que « $\forall(T : *)$ » / « $\exists(T : *)$ » (puisque c'est la seule forme de quantification autorisée par le système F).

À titre d'exemple, dans le système F, on peut écrire le terme $\Lambda A.\, \Lambda B.\, \lambda(x : A).\, \lambda(y : B).\, x$, qui prouve (ou a pour type) $\forall A.\, \forall B.\, (A \Rightarrow B \Rightarrow A)$, ou encore $\Lambda A.\, \lambda(f : \forall B.B).\, fA$ qui prouve (ou a pour type) $\forall A.\, ((\forall B.\, B) \Rightarrow A)$.

(1) Montrer (en donnant des λ-termes) dans le système F que $\forall A.\, (A \Rightarrow \forall C.((A \Rightarrow C) \Rightarrow C))$ et que $\forall A.\, ((\forall C.((A \Rightarrow C) \Rightarrow C)) \Rightarrow A)$. Si on voit ces termes comme des programmes ayant les types en question, décrire brièvement ce que font ces programmes.

(2) Montrer de même que $\forall A.\, \forall B.\, (A \wedge B \Rightarrow \forall C.((A \Rightarrow B \Rightarrow C) \Rightarrow C))$ et que $\forall A.\, \forall B.\, ((\forall C.((A \Rightarrow B \Rightarrow C) \Rightarrow C)) \Rightarrow A \wedge B)$. Quel rapport avec l'exercice 2.4 ?

(3) La question précédente indique que dans le système F, $P \wedge Q$ peut être considéré comme un synonyme de $\forall C.\, (P \Rightarrow Q \Rightarrow C) \Rightarrow C$ (ces deux propositions sont équivalentes ; ceci ne montre pas tout à fait qu'on peut se dispenser entièrement du signe $\wedge$ car il n'est pas évident qu'on n'en ait pas besoin dans les démonstrations, mais il s'avère que c'est le cas). Proposer de façon semblable un synonyme de $P \vee Q$ ne faisant intervenir que $\Rightarrow$ et $\forall$, et montrer l'équivalence.

(4) Proposer une piste pour réécrire $\exists T.\, Q$ (où T peut apparaître libre dans Q) en ne faisant intervenir que $\Rightarrow$ et $\forall$. (Comme le système F ne permet pas de quantifier sur les propositions dépendant d'une autre proposition [2], on se contentera d'écrire les équivalences sur un Q quelconque.)

Exercice 5.2. ($\star$)

(1) Montrer chacune des propositions suivantes en pure logique du premier ordre (où A, B désignent des relations unaires) en donnant un λ-terme de preuve : **(a)** $(\forall x.A(x) \Rightarrow B(x)) \Rightarrow (\forall x.A(x)) \Rightarrow (\forall x.B(x))$ **(b)** $(\forall x.A(x) \Rightarrow B(x)) \Rightarrow (\exists x.A(x)) \Rightarrow (\exists x.B(x))$ **(c)** $(\exists x.A(x) \Rightarrow B(x)) \Rightarrow (\forall x.A(x)) \Rightarrow (\exists x.B(x))$

(2) En interprétant chacun de ces termes comme un programme, en en oubliant les types tels qu'on vient de les écrire, qu'obtient-on si on demande à OCaml (c'est-à-dire en fait à l'algorithme de Hindley-Milner étendu avec des types produits) de leur reconstruire des types ? Commenter brièvement.

6 Arithmétique du premier ordre et théorème de Gödel

Exercice 6.1. ($\star$)

Démontrer dans l'arithmétique de Heyting que $\forall n.(0 + n = n)$.

2. Il faudrait pour cela faire apparaître le type $* \to *$ et quantifier dessus, ce qui dépasse le système F (le système appelé « Fω » le permet).

Exercice 6.2. ($\star\star\star\star$)

Soit T une théorie logique telle que [3] l'arithmétique de Heyting HA, l'arithmétique de Peano PA, Coq ou ZFC. On se propose ici de démontrer le **théorème de Löb** : si A est un énoncé de T et si T prouve « si T prouve A, alors A », alors T prouve A.

On construit pour cela le programme g_A suivant : g_A cherche une preuve dans T de l'énoncé « si g_A termine, alors A », et s'il en trouve une, il termine immédiatement.

(1) Expliquer pourquoi g_A est bien défini. A-t-on besoin d'arriver à tester la véracité ou la démontrabilité de A pour le construire ?

(2) Supposons que g_A termine : montrer que A est démontrable dans T.

(3) Remarquer que le point précédent est lui-même démontrable dans T.

(4) Déduire de (3) que si T prouve « si T prouve A, alors A », alors g_A termine.

(5) Déduire de (2) et (4) que si T prouve « si T prouve A, alors A », alors T provue A.

Exercice 6.3. ($\star\star\star\star$)

Soit T une théorie logique telle que [4] l'arithmétique de Heyting HA, l'arithmétique de Peano PA, Coq ou ZFC.

On considère le programme h suivant : h cherche une démonstration dans T du fait que h termine, et s'il en trouve une, il termine immédiatement.

(1) Expliquer pourquoi h est bien défini.

(Remarquez que ce programme est « conciliant » : alors que dans la démonstration du théorème de Gödel on a considéré un programme « contrariant » qui fait le contraire de la démonstration qu'il a trouvée, ici, si le programme trouve une preuve de son arrêt, il obéit sagement.)

On va prouver que h termine effectivement.

(2) Montrer que si T prouve que h termine, alors h termine.

(3) Remarquer que le point précédent est lui-même démontrable dans T.

(4) Considérer l'énoncé « le programme h termine », appliquer le théorème de Löb énoncé à l'exercice 6.2, et conclure.

Remarque : on peut résumer la conclusion de cet exercice en disant que « ceci est un théorème » est un théorème (mais la preuve, comme on vient de le voir, n'est pas vraiment évidente).

Exercice 6.4. ($\star\star\star\star$)

Dans cet exercice, on s'intéresse au programme p suivant [5] : il prend en entrée un entier qu'il ignore ; il part de $N = 42$, puis il énumère toutes les démonstrations dans l'arithmétique de Peano dont la longueur (en nombre de symboles sur un alphabet fini fixé) vaut au plus $10^{10^{100}}$, et, pour chacune, si la *conclusion* de la démonstration est une affirmation du type « le e-ième programme termine sur l'entrée i » (i.e., $\varphi_e(i)\downarrow$) avec e et i des entiers naturels explicites, alors il exécute $\varphi_e(i)$ et si ce programme termine, il ajoute le résultat (considéré comme un entier naturel) à N. Une fois que tout ceci est fait, il renvoie N.

(1) Commenter brièvement les aspects suivants du programme p : est-il très long à écrire ? est-il plus ou moins évident qu'il termine ?

3. Comme pour le théorème de Gödel, il s'agit essentiellement que T soit codable par des entiers naturels, permette de formaliser l'arrêt des machines de Turing, et ait des démonstrations algorithmiquement testables. On ne donnera pas de conditions exactes (ce serait trop fastidieux) mais les théories proposées en exemple les vérifient.

4. Voir la note 3.

5. On pourra supposer ici les programmes écrits dans un langage de raisonnablement haut niveau.

On désigne par « $\Sigma_1\mathsf{Sound}(\mathsf{PA})$ » (peu importe ce que signifie « $\Sigma_1\mathsf{Sound}$ » ici) l'affirmation suivante : « si l'arithmétique de Peano prouve $\varphi_e(i)\!\downarrow$, alors effectivement $\varphi_e(i)\!\downarrow$ ». On ne cherchera pas (pour l'instant) à se demander si $\Sigma_1\mathsf{Sound}(\mathsf{PA})$ est vrai, ou bien démontrable.

(2) En *admettant* l'affirmation $\Sigma_1\mathsf{Sound}(\mathsf{PA})$ qu'on vient de définir, montrer que le programme p termine en temps fini. Expliquer, de plus, pourquoi l'arithmétique de Peano PA prouve ce fait.

(3) Sans entrer dans les détails, et toujours en admettant $\Sigma_1\mathsf{Sound}(\mathsf{PA})$, expliquer pourquoi la valeur renvoyée par p (i.e., $\varphi_p(0)$) est gigantesque, par exemple beaucoup *beaucoup* plus grande que $10^{10^{10^{1000}}}$ itérations de la fonction $n \mapsto A(n,n,n)$, où A est la fonction d'Ackermann, en partant de $10^{10^{10^{1000}}}$ (*indication :* esquisser comment on pourrait écrire un programme q qui calcule cette dernière valeur, et comment on prouverait que ce programme q termine). Quelle est la partie la plus longue dans l'exécution de p : l'énumération des démonstrations ou l'exécution des programmes dont on a prouvé l'arrêt ?

(4) Toujours en admettant $\Sigma_1\mathsf{Sound}(\mathsf{PA})$, montrer que la démonstration la plus courte de l'arrêt de p dans l'arithmétique de Peano est de longueur supérieure à $10^{10^{100}}$.

En fait, ZFC prouve l'affirmation $\Sigma_1\mathsf{Sound}(\mathsf{PA})$ énoncée plus haut, et la démonstration n'est pas terriblement longue (on ne rentre pas dans les détails ici).

(5) Conclure que la démonstration du fait que p termine, bien que démontrable dans l'arithmétique de Peano, est beaucoup plus long à démontrer que dans ZFC.

7 Divers

Exercice 7.1. ($\star\star\star\star\star$)

Vous êtes dans un donjon. Devant vous se trouvent trois portes, étiquetées A, B, C. Derrière l'une de ces trois portes, mais vous ne savez pas laquelle, se trouve un dragon, qui vous dévorera si vous l'ouvrez ; les deux autres portes, qu'on appellera « sûres » dans la suite mènent à la sortie du donjon (avec un trésor à la clé). Votre but est d'ouvrir une porte sûre (peu importe laquelle).

Sur chaque porte est affiché un programme (p_A, p_B, p_C) qui constitue un indice pour trouver une porte sûre. Plus précisément, le programme affiché sur chaque porte prend une entrée q et va, *sous certaines conditions* terminer et renvoyer l'étiquette d'une des deux autres portes qui soit sûre. Plus exactement, si on suppose que X est l'étiquette de la porte avec le dragon :

— Le programme p_X qui est affiché sur la porte au dragon va *toujours* terminer (quelle que soit l'entrée q qu'on lui a passée) et renvoyer l'étiquette d'une des deux autres portes Y, Z (qui sont toutes les deux sûres puisqu'il n'y a qu'un seul dragon) : $\varphi_{p_X}(q)\!\downarrow \in \{Y, Z\}$ quel que soit q (mais noter que le résultat peut dépendre de q).

— Le programme p_Y qui est affiché sur une porte sûre Y va terminer et renvoyer l'étiquette de l'autre porte sûre Z, mais *à condition* qu'on lui ait passé comme argument un programme q (sans argument) qui fasse exactement ça (i.e., à condition que q lui-même termine et renvoie Z) : si $\varphi_q(0)\!\downarrow = Z$ alors [6] $\varphi_{p_Y}(q)\!\downarrow = Z$ (dans tout autre cas, on n'a aucune garantie sur $\varphi_{p_Y}(q)$: il pourrait ne pas terminer, renvoyer une étiquette fausse, ou renvoyer complètement autre chose).

Comment pouvez-vous utiliser ces indications pour ouvrir (de façon certaine, et même algorithmique d'après p_A, p_B, p_C) une porte sûre ?

(*Indication :* on pourra utiliser l'exercice 1.16(1).)

6. Le 0 est mis pour une absence d'argument.

Question subsidiaire (indépendante) : montrer que l'énigme ci-dessus serait insoluble si au lieu d'avoir des *programmes* affichés sur les portes on avait des tableaux de valeurs (tableaux donnant un résultat « A », « B », « C » ou n'importe quoi en fonction d'une entrée « A », « B » ou « C ») avec les mêmes contraintes (i.e. : le tableau sur la porte du dragon donne l'étiquette d'une des deux autres portes quelle que soit l'entrée, et le tableau sur une porte sûre donne l'étiquette de l'autre porte sûre si l'entrée est justement l'étiquette de l'autre porte sûre).

Exercice 7.2. ($\star\star\star\star\star$)

On considère la formule propositionnelle suivante (due à V. Plisko) :

$$\begin{aligned}
&\Big((\neg\neg A \Leftrightarrow \neg B \wedge \neg C) \wedge (\neg\neg B \Leftrightarrow \neg C \wedge \neg A) \wedge (\neg\neg C \Leftrightarrow \neg A \wedge \neg B) \\
&\quad \wedge \big((\neg A \Rightarrow (\neg B \vee \neg C)) \Rightarrow (\neg B \vee \neg C) \big) \\
&\quad \wedge \big((\neg B \Rightarrow (\neg C \vee \neg A)) \Rightarrow (\neg C \vee \neg A) \big) \\
&\quad \wedge \big((\neg C \Rightarrow (\neg A \vee \neg B)) \Rightarrow (\neg A \vee \neg B) \big) \Big) \\
\Rightarrow\ &\big(\neg A \vee \neg B \vee \neg C \big)
\end{aligned}$$

(on notera la symétrie entre A, B, C qui rend la formule moins complexe qu'elle n'en a l'air).

Déduire de l'exercice 7.1 que cette formule est réalisable mais (d'après la question subsidiaire de cet exercice) pas Medvedev-valide.

(*Indication :* comme A, B, C n'apparaissent qu'avec une négation partout dans cette formule, leur contenu n'a pas d'importance, la seule chose qui importe est qu'ils aient ou non des éléments — ou dans le cas des problèmes finis, des solutions ; il s'agit alors de reconnaître que la formule reflète exactement les termes de l'énigme.)