Logique et Fondements de l'Informatique
Exercices corrigés

David A. Madore

12 novembre 2024

INF1110

Git : `10c0912 Fri Jan 26 18:53:33 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.

Corrigé. On peut calculer $f(n)$ selon l'algorithme suivant : calculer $N = 10^n$, puis pour i allant de 0 à $2N$, tester si $i^2 \leq 2N^2 < (i+1)^2$: lorsque c'est le cas (et ce sera le cas pour exactement un i dans l'intervalle), renvoyer le reste $i\%10$ de la division euclidienne de i par 10.

Cet algorithme est correct car l'inégalité $i^2 \leq 2N^2 < (i+1)^2$ testé équivaut à $\frac{i}{N} \leq \sqrt{2} < \frac{i+1}{N}$, ce qui se produit pour exactement un i, à savoir $\lfloor \sqrt{2} \times 10^n \rfloor$ (on peut arrêter la boucle à $2N$ car $\sqrt{2} < 2$), et que le dernier chiffre décimal $i\%10$ de ce i est le n-ième chiffre de l'écriture décimale de $\sqrt{2}$.

D'autre part, comme on a donné un algorithme explicite, f est calculable. Mieux : comme la boucle utilisée est bornée *a priori*, f est primitive récursive. ✓

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 ?

Corrigé. La réponse est non dans les deux cas : pour le voir appelons $H := \{e \in \mathbb{N} : \varphi_e(0)\downarrow\}$ (disons) : il est indécidable par une des variations du problème de l'arrêt (ou par le théorème de Rice). Le fait que $H \subseteq \mathbb{N}$ avec $\mathbb{N}$ décidable réfute (1), et le fait que $\varnothing \subseteq H$ avec $\varnothing$ décidable réfute (2). ✓

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 $f : \mathbb{N} \to \mathbb{N}$ totale calculable et strictement croissante. Montrer que l'image $f(\mathbb{N})$ (c'est-à-dire $\{f(i) : i \in \mathbb{N}\}$) est décidable.

Corrigé. **(1)** L'algorithme évident suivant semi-décide $\{f(i) : i \in \mathbb{N}\}$: donné $m \in \mathbb{N}$ l'entier à tester, faire une boucle infinie sur i parcourant les entiers naturels et pour chacun, tester si $f(i) = m$: si c'est le cas, terminer et répondre « oui », sinon, continuer la boucle.

(2) L'algorithme évident suivant décide $\{f(i) : i \in \mathbb{N}\}$: donné $m \in \mathbb{N}$ l'entier à tester, faire une boucle pour i parcourant les entiers naturels, et pour chacun, tester si $f(i) = m$: si c'est le cas, terminer et répondre « oui », tandis que si $f(i) > m$, terminer et répondre « non », sinon, continuer la boucle. La boucle termine en temps fini car $f(i) \geq i$ (inégalité claire pour une fonction $\mathbb{N} \to \mathbb{N}$ strictement croissante) et notamment la boucle s'arrêtera au pire lorsque i vaut $m + 1$. (Du coup, si on préfère, on peut réécrire la boucle potentiellement infinie comme une boucle pour i allant de 0 à m.) ✓

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.

Corrigé. L'ensemble F des fonctions partielles calculables $f : \mathbb{N} \dashrightarrow \mathbb{N}$ telles que $f(0) = f(1)$ n'est ni vide (la fonction totale constante de valeur 0 est dans F) ni plein (la fonction totale identité n'est pas dans F). D'après le théorème de Rice, l'ensemble des e tels que $\varphi_e^{(1)} \in F$ est indécidable : c'est exactement ce qui était demandé. ✓

Exercice 1.5. ($\star\star\star$)

(1) Soit $B \subseteq \mathbb{N}$ semi-décidable et non-vide. Montrer qu'il existe $f : \mathbb{N} \to \mathbb{N}$ 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 $\tilde{f} : \mathbb{N}^2 \to \mathbb{N}$ 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 $f : \mathbb{N} \dashrightarrow \mathbb{N}$ 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 ».)

Corrigé. **(1)** La fonction $\tilde{f} : \mathbb{N}^2 \to \mathbb{N}$ définie dans l'indication est calculable (et d'ailleurs même primitive récursive) : si on a pris la définition avec T le fait que T soit p.r. fait partie du théorème de la forme normale ; si on préfère les machines de Turing, c'est le fait qu'on peut simuler l'exécution de $\mathscr{M}$ pour n étapes (de façon p.r.). Et on voit qu'on a $\tilde{f}(n, m) \in B$ dans tous les cas : donc $\tilde{f}(\mathbb{N}^2) \subseteq B$. Mais réciproquement, si $m \in B$, alors $\varphi_e(m) \downarrow$ (si on préfère les machines de Turing, $\mathscr{M}$ termine sur l'entrée m), et ceci dit précisément qu'il existe n tel que $\tilde{f}(n, m) = m$, donc $m \in \tilde{f}(\mathbb{N}^2)$; bref, $B \subseteq \tilde{f}(\mathbb{N}^2)$. On a donc $\tilde{f}(\mathbb{N}^2) = B$ par double inclusion. Quitte à remplacer $\tilde{f} : \mathbb{N}^2 \to \mathbb{N}, (n, m) \mapsto \tilde{f}(n, m)$ par $f : \mathbb{N} \to \mathbb{N}, \langle n, m \rangle \mapsto \tilde{f}(n, m)$, on a $f(\mathbb{N}) = B$.

(2) Ici on ne peut pas appliquer bêtement l'algorithme exposé dans l'exercice 1.3 question (1) car si le calcul de $f(i)$ ne termine pas, il bloquera tous les suivants. Il faut donc mener le calcul des $f(i)$ « en parallèle ». On va procéder par énumération des couples (n, i) et lancer le calcul de $f(i)$ sur n étapes.

Plus précisément : considérons l'algorithme suivant : il prend en entrée un entier m dont il s'agit de semi-décider s'il appartient à $f(\mathbb{N})$. L'algorithme fait une boucle infinie sur p parcourant les entiers naturels : chaque p est d'abord décodé comme le code $\langle n, i \rangle$ d'un couple d'entiers naturels (ceci est bien sûr calculable). On teste si l'exécution de $f(i)$ termine en $\leq n$ étapes (ou, si on préfère le théorème de la forme de normale, on teste si $T(n, e, \langle\langle i \rangle\rangle)$, où e est un code de la fonction $f = \varphi_e^{(1)}$) : si oui, et si la valeur $f(i)$ calculée est égale à l'entier m considéré, on termine en renvoyant « oui », sinon on continue la boucle.

Cet algorithme semi-décide bien $f(\mathbb{N})$: en effet, dire que $m \in f(\mathbb{N})$, équivaut à l'existence de i tel que $f(i)\downarrow = m$, c'est-à-dire à l'existence de n, i tel que l'algorithme renverra « oui » en testant $\langle n, i \rangle$.

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 $U(n)$).

(*Variante :* plutôt qu'utiliser le codage des couples $\langle n, i\rangle$, on peut aussi faire ainsi : on parcourt les entiers naturels p en une boucle infini et pour chacun on effectue deux boucles bornées pour $0 \leq n \leq p$ et $0 \leq i \leq p$: peu importent les bornes précises, l'important est que pour p assez grand on va finir par tester le couple (n, i).) ✓

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.

Corrigé. On notera « φ » pour « $\varphi^{(1)}$ » de manière à alléger les notations.

(1) L'ensemble des fonctions calculables $\mathbb{N} \dashrightarrow \mathbb{N}$ qui sont en fait totales ($\mathbb{N} \to \mathbb{N}$) n'est ni vide (la fonction totale constante de valeur 0 est dedans) ni plein (la fonction nulle part définie n'est pas dedans). D'après le théorème de Rice, l'ensemble T des e tels que φ_e soit totale est indécidable : c'est exactement ce qui était demandé.

(2) Toujours d'après le théorème de Rice, ou comme variante du problème de l'arrêt (qui s'y ramène par le théorème s-m-n), l'ensemble H n'est pas décidable. Il est cependant semi-décidable par universalité (on peut lancer l'exécution de e sur l'entrée 0 et, si elle termine, renvoyer « oui »). On en déduit que $\complement H$ n'est pas semi-décidable (car si H et $\complement H$ étaient semi-décidables, H serait décidable, ce qu'il n'est pas).

(3) Considérons la fonction ρ qui prend en entrée un programme d (supposé d'un argument) et renvoie le programme $e =: \rho(d)$ (toujours d'un argument) qui ignore son argument et exécute d sur l'entrée 0 : essentiellement par le théorème s-m-n, cette fonction ρ est totale calculable (d'ailleurs même p.r.). Par définition, on a $\varphi_{\rho(d)}(n) = \varphi_d(0)$ (rappelons que ceci signifie que chacun est défini ssi l'autre l'est et, le cas échéant, que ces valeurs sont égales). Notamment, si $\varphi_d(0) \downarrow$, alors $\varphi_{\rho(d)}$ est totale (et constante !), tandis que si $\varphi_d(0) \uparrow$, alors $\varphi_{\rho(d)}$ n'est nulle part définie (donc certainement pas totale).

Bref, on a construit $\rho\colon \mathbb{N} \to \mathbb{N}$ totale calculable telle que $d \in H$ si et seulement si $\rho(d) \in T$, ou, ce qui revient au même, $d \in \complement H$ si et seulement si $\rho(d) \in \complement T$. En termes de réductions, ceci signifie $H \leq_{\mathrm{m}} T$, ou, ce qui revient au même, $\complement H \leq_{\mathrm{m}} \complement T$ (le symbole « $\leq_{\mathrm{m}}$ » désignant la réduction many-to-one). Comme $\complement H$ n'est pas semi-décidable, $\complement T$ ne l'est pas non plus.

Remarque : On n'est pas obligé d'utiliser le terme de « réduction many-to-one » pour argumenter que $\complement T$ n'est pas semi-décidable : on peut simplement dire « supposant par l'absurde que $\complement T$ soit semi-décidable, on pourrait semi-décider $\complement H$ de la façon suivante : donné d, on calcule $\rho(d)$, on semi-décide si $\rho(d) \in \complement T$ et, si c'est le cas, on termine en renvoyant "oui" ; or ce n'est pas possible, d'où une contradiction ».

(4) Considérons la fonction σ qui prend en entrée un programme d (supposé d'un argument) et renvoie le programme $e := \sigma(d)$ (toujours d'un argument) défini ainsi : le programme e prend en entrée un nombre n et exécute le programme d sur l'entrée 0 pendant $\leq n$ étapes (mettons que ce soient des machines de Turing, sinon remplacer cet argument par une recherche d'arbre de calcul parmi les entiers naturels de 0 à n) : si cette exécution a terminé en $\leq n$ étapes, alors d effectue une boucle infinie, sinon d termine (et renvoie, disons, 1729).

Il n'y a pas de difficulté à coder ce programme e (on rappelle qu'exécuter un programme donné sur $\leq n$ étapes est calculable, d'ailleurs même primitif récursif), et de plus la fonction σ transformant d en e est elle-même calculable (et d'ailleurs elle aussi primitive récursive).

Par définition de $e := \sigma(d)$, la fonction φ_e est :

— soit définie pour tout n (et de valeur 1729), ce qui se produit exatement lorsque l'exécution de d ne termine jamais, i.e. $\varphi_d(0) \uparrow$,

— soit définie jusqu'en un certain n et non définie après, ce qui se produit exactement lorsque l'exécution de d termine en un certain nombre d'étapes, i.e. $\varphi_d(0) \downarrow$.

En particulier, si $\varphi_d(0) \uparrow$, alors $\varphi_{\sigma(d)}$ est totale (et constante !), tandis que si $\varphi_d(0) \downarrow$, alors $\varphi_{\sigma(d)}$ n'est pas totale.

Bref, on a construit $\sigma : \mathbb{N} \to \mathbb{N}$ totale calculable telle que $d \in H$ si et seulement si $\sigma(d) \notin T$, ou, ce qui revient au même, $d \in \complement H$ si et seulement si $\sigma(d) \in T$. En termes de réductions, ceci signifie $\complement H \leq_{\mathrm{m}} T$. Comme $\complement H$ n'est pas semi-décidable, T ne l'est pas non plus.

Remarque : Comme dans la question précédente, on n'est pas obligé d'utiliser le terme de « réduction many-to-one » pour argumenter que T n'est pas semi-décidable : on peut simplement dire « supposant par l'absurde que T soit semi-décidable, on pourrait semi-décider $\complement H$ de la façon suivante : donné d, on calcule $\sigma(d)$, on semi-décide si $\sigma(d) \in T$ et, si c'est le cas, on termine en renvoyant "oui" ; or ce n'est pas possible, d'où une contradiction ». ✓

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

Soit $f : \mathbb{N} \to \mathbb{N}$ 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).)

Corrigé. Montrons que (1) implique (2) : si on dispose d'un algorithme $\mathscr{F}$ capable de calculer $f(i)$ en fonction de i, alors il est facile d'écrire un algorithme $\mathscr{D}$ capable de décider si $q = f(i)$ (il suffit de calculer $f(i)$ avec l'algorithme $\mathscr{F}$ supposé exister, de comparer avec la valeur de q fournie, et de renvoyer vrai/1 si elles sont égales, et faux/0 sinon), c'est-à-dire que l'algorithme $\mathscr{D}$ décide Γ_f.

Le fait que (2) implique (3) est évident car tout ensemble décidable est en particulier semi-décidable.

Montrons que (2) implique (1) même si ce ne sera au final pas utile : supposons qu'on ait un algorithme $\mathscr{D}$ qui décide Γ_f (i.e., donnés (i, q), termine toujours en temps fini, en répondant « oui » si $q = f(i)$ et « non » si $q \neq f(i)$), et on cherche à écrire un algorithme $\mathscr{F}$ qui calcule $f(i)$. Pour cela, donné un i, il suffit de lancer l'algorithme $\mathscr{D}$ successivement sur les valeurs $(i, 0)$ puis $(i, 1)$ puis $(i, 2)$ et ainsi de suite (c'est-à-dire faire une boucle infinie sur q parcourant les entiers naturels et lancer $\mathscr{D}$ sur chaque couple (i, q)) jusqu'à trouver un q pour lequel $\mathscr{D}$ réponde vrai : on termine alors en renvoyant la valeur q qu'on a trouvée, qui vérifie $q = f(i)$ par définition de $\mathscr{D}$. L'algorithme $\mathscr{F}$ qu'on vient de décrire termine toujours car f était supposée totale, donc il existe bien un q pour lequel $\mathscr{D}$ répondra « oui ».

Reste à montrer que (3) implique (1) : supposons maintenant qu'on ait un algorithme $\mathscr{S}$ qui « semi-décide » Γ_f (i.e., donnés (i, q), termine en temps fini et répond « oui » si $q = f(i)$, et ne termine pas sinon), et on cherche à écrire un algorithme qui, donné i en entrée, calcule $f(i)$. Notre algorithme (appelons-le $\mathscr{F}$) fait une boucle infinie sur p parcourant les entiers naturels : chaque p est d'abord décodé comme le code $\langle n, q\rangle$ d'un couple d'entiers naturels. On teste si l'exécution de $\mathscr{S}$ sur l'entrée (i, q) termine en $\leq n$ étapes (ce qui est bien faisable algorithmiquement) : si oui, on renvoie la valeur q ; sinon, on continue la boucle.

Cet algorithme $\mathscr{F}$ termine toujours : en effet, pour chaque i donné, il existe q tel que $(i, q) \in \Gamma_f$, à savoir $q = f(i)$; et alors l'algorithme $\mathscr{S}$ doit terminer sur l'entrée (i, q), c'est-à-dire que pour n assez grand, il termine en $\leq n$ étapes, donc $\mathscr{F}$ terminera lorsqu'il arrivera à $p = \langle n, q\rangle$, et il renverra bien q comme annoncé. On a donc montré que f était calculable puisqu'on a exhibé un algorithme qui la calcule.

(Comme dans l'exercice 1.5, on peut utiliser le T de la forme normale de Kleene au lieu de parler d'« étapes » d'exécution d'une machine de Turing. Aussi, plutôt qu'utiliser le codage des couples $\langle n, i\rangle$, on peut préférer faire ainsi : on parcourt les entiers naturels p en une boucle infini et pour chacun on effectue deux boucles bornées pour $0 \leq n \leq p$ et $0 \leq q \leq p$: peu importent les bornes précises, l'important est que pour p assez grand on va finir par tester le couple (n, q).) ✓

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

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

$$N := \{e \in \mathbb{N} : \exists e' \in \mathbb{N}.(\psi_{e'}^{(1)} \text{ définie et } \varphi_e^{(1)} = \psi_{e'}^{(1)})\}$$

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.

Corrigé. L'ensemble M est l'ensemble des codes valables de fonction primitives récursives, c'est-à-dire de codes légitimes dans le langage primitif récursif ; l'ensemble N qui est $\{e \in \mathbb{N} : \varphi_e^{(1)} \text{ est p.r.}\}$ est l'ensemble des codes de fonctions générales récursives qui s'avèrent être primitives récursives (même si ce n'est pas forcément manifeste sur le programme). Si on préfère, M est l'ensemble des *intentions* primitives récursives, alors que N est l'ensemble des intentions dont l'*extension* est primitive récursive ; *grosso modo*, l'appartenance à M se lit sur le code de la fonction, celle à N se lit sur les valeurs de la fonction.

Manifestement, $M \subseteq N$, car si $\psi_e^{(1)}$ est définie, on a $\varphi_e^{(1)} = \psi_e^{(1)}$ (la définition des fonctions générales récursives *étend* celle des fonctions p.r.). L'inclusion dans l'autre sens ne vaut pas : on peut calculer une fonction non p.r., jeter le résultat, et renvoyer 0, ce qui fournit un code e tel que $\varphi_e^{(1)}$ est primitive récursive (donc $e \in N$) et pourtant $\psi_e^{(1)}$ n'est pas définie (donc $e \notin M$).

L'ensemble M est décidable : on peut décider de façon algorithmique si e est un numéro valable de fonction primitive récursive (i.e., si $\psi_e^{(1)}$ est définie), il s'agit pour cela simplement de « décoder » e et de vérifier qu'il suit les conventions utilisées pour numéroter les fonctions primitives récursives (pour être très précis, le décodage termine parce que le code d'une liste est supérieur à tout élément de cette liste).

L'ensemble N n'est pas décidable : si F désigne l'ensemble des fonctions p.r. $\mathbb{N} \to \mathbb{N}$ (c'est-à-dire l'image de M par $e \mapsto \psi_e^{(1)}$), alors N est $\{e \in \mathbb{N} : \varphi_e^{(1)} \in F\}$, et comme F n'est ni vide ni l'ensemble de toutes les fonctions générales récursives, le théorème de Rice dit exactement que N est indécidable. ✓

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

On considère la fonction $f : \mathbb{N}^2 \to \mathbb{N}$ qui à (e, x) associe 1 si $\psi_e^{(1)}(x) = 0$, et 0 sinon (y compris si $\psi_e^{(1)}$ n'est pas définie) ; ici, $e \mapsto \psi_e^{(1)}$ 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 ?

Corrigé. La fonction f est calculable. En effet,

- on peut décider de façon algorithmique si e est un numéro valable de fonction primitive récursive (i.e., si $\psi_e^{(1)}$ est définie), il s'agit pour cela simplement de « décoder » e et de vérifier qu'il suit les conventions utilisées pour numéroter les fonctions primitives récursives (pour être très précis, le décodage termine parce que le code d'une liste est supérieur à tout élément de cette liste) ;
- lorsque c'est le cas, on peut calculer $\psi_e^{(1)}(x)$ car quand elle est définie elle coïncide avec $\varphi_e^{(1)}(x)$ (numérotation des fonctions générales récursives), dont on sait qu'il est calculable (universalité) ;
- calculer f ne pose ensuite aucune difficulté.

Montrons que f n'est pas primitive récursive (on a vu en cours que $(e, x) \mapsto \psi_e^{(1)}(x)$ ne l'est pas, mais cela ne suffit pas : on pourrait imaginer que le fait qu'il soit égal à 0 soit plus facile à tester). Pour cela, supposons par l'absurde que f soit primitive récursive. Par le théorème de récursion de Kleene, il existe e tel que $\psi_e^{(1)}(x) = f(e, x)$. Or la définition même de f fait que $f(e, x) \neq \psi_e^{(1)}(x)$ dans tous les cas : ceci est une contradiction. Donc f n'est pas primitive récursive.

Cela ne change bien sûr rien d'échanger 0 et 1, c'est-à-dire de remplacer f par $1 - f$ (l'une est récursive, resp. p.r., ssi l'autre l'est), mais la démonstration ne se serait pas appliquée telle quelle. ✓

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

Soit

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

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

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

Corrigé. Comme dans le début du corrigé de l'exercice 1.9, on explique qu'on peut décider si $\psi_e^{(1)}(n) \downarrow$ (il s'agit juste de vérifier si e est un code valable de fonction p.r.) et, une fois ce point vérifié, si $\psi_e^{(1)}(n) = 0$ (on peut calculer $\psi_e^{(1)}(n) = \varphi_e^{(1)}(n)$ par universalité des fonctions générales récursives).

Dès lors, pour semi-décider si $e \in Z$, il suffit de faire une boucle infinie pour n parcourant les entiers naturels, décider si $\psi_e^{(1)}(n) = 0$ pour chacun, et si l'un d'eux est effectivement nul, terminer et renvoyer « oui », sinon on continue la boucle. Ceci montre que Z est semi-décidable.

Montrons qu'il n'est pas décidable : pour cela, on va ramener le problème de l'arrêt à Z. C'est en fait essentiellement ce que fait le théorème de la forme normale de Kleene : en effet, considérons (p, x) dont il s'agit de décider si $\varphi_p^{(1)}(x) \downarrow$: d'après le théorème de la forme normale, ceci se produit si et seulement si il existe un (entier codant un) arbre de calcul n attestant que $\varphi_p^{(1)}(x) \downarrow$, ce qu'on écrit $T(n, p, \langle\langle x \rangle\rangle)$, où T est un prédicat p.r., c'est-à-dire qu'il s'écrit $t(n, p, \langle\langle x \rangle\rangle) = 0$ pour une certaine fonction p.r. t (qui teste si n code un arbre de calcul valable pour $\varphi_p^{(1)}(x)$ et renvoie 0 si c'est le cas, 1 sinon). On a ainsi $\varphi_p^{(1)}(x) \downarrow$ ssi $\exists n \in \mathbb{N}.\, (t(n, p, \langle\langle x \rangle\rangle) = 0)$. Maintenant, d'après le théorème s-m-n, on peut calculer de façon p.r. en p et x le code $\rho(p, x)$ d'une fonction p.r. telle que $\psi_{\rho(p,x)}^{(1)}(n) = t(n, p, \langle\langle x \rangle\rangle)$, et d'après ce qui a été dit juste avant, on a $\rho(p, x) \in Z$, c'est-à-dire $\exists n \in \mathbb{N}.\, (t(n, p, \langle\langle x \rangle\rangle) = 0)$, se produit si et seulement si $\varphi_p^{(1)}(x) \downarrow$, c'est-à-dire $(p, x) \in \mathscr{H}$ (où $\mathscr{H} := \{(p, x) \in \mathbb{N}^2 : \varphi_p^{(1)}(x) \downarrow\}$ désigne le problème de l'arrêt). Ceci constitue une réduction *many-to-one* de $\mathscr{H}$ à Z, donc Z ne peut pas être décidable : en effet, si Z était décidable, pour tester si $(p, x) \in \mathscr{H}$ il suffirait de tester si $\rho(p, x) \in Z$, donc le problème de l'arrêt serait décidable, ce qui n'est pas le cas.

(De nouveau, si on n'aime pas le théorème de la forme normale de Kleene, on peut faire ça avec des étapes de machine de Turing : appeler $t(n, p, x)$ la fonction qui renvoie 0 si la machine de Turing codée par p termine en $\leq n$ étapes à partir de la configuration initiale codée par x, et 1 sinon : le reste du raisonnement est essentiellement identique.) ✓

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.)

Corrigé. **(1)** Commençons par remarquer que « passer par un nombre fini de configurations distinctes » équivaut à « terminer ou revenir à une configuration déjà atteinte ». En effet, dans un sens, si l'exécution termine (i.e., termine en temps fini), il est évident qu'elle n'a parcouru qu'un nombre fini de configurations distinctes ; mais si elle revient à une configuration déjà atteinte, la machine boucle indéfiniment à partir de cet état puisque l'exécution est déterministe (la configuration contient toute l'information nécessaire à l'exécution de la machine M) : si $C^{(i)} = C^{(j)}$ avec $i < j$ alors $C^{(i+k)} = C^{(j+k)}$ pour tout k, et donc toute configuration atteinte est une de $C^{(0)}, \ldots, C^{(j)}$) Dans l'autre sens, si la machine ne passe que par un nombre fini de configurations distinctes et ne s'arrête pas, par le principe des tiroirs, il y

aura forcément une configuration atteinte plusieurs fois, c'est-à-dire $C^{(i)} = C^{(j)}$ avec $i < j$. Ceci montre l'équivalence affirmée.

Pour semi-décider $\mathscr{F}$, il suffit de lancer l'exécution à partir de C_0, en enregistrant chaque configuration atteinte (on rappelle qu'une configuration est une donnée finie puisqu'il n'y a, à un instant donné, qu'un nombre fini de 1 sur la bande), et la comparer à toutes les configurations précédemment atteintes. Si on repasse dans une configuration déjà atteinte, on termine et répond « oui », sinon on continue l'exécution. D'après ce qui vient d'être dit, ceci semi-décide $\mathscr{F}$.

(2) Supposons par l'absurde qu'on soit capable de décider $\mathscr{F}$ et montrons que ceci permettrait de décider le problème de l'arrêt à partir de la configuration vierge (dont on a vu en cours qu'il est indécidable). En effet, donné M, on commence par tester (grâce à notre hypothèse) si $M \in \mathscr{F}$: si ce n'est pas le cas, on sait déjà que M ne terminera pas et on répond « non » ; si c'est le cas, on sait que l'exécution de M à partir de la configuration vierge conduira soit à l'arrêt soit à retomber sur une configuration déjà atteinte : il suffit de simuler cette exécution en enregistrant chaque configuration atteinte, et, si on tombe sur une configuration déjà atteinte on répond « non », tandis que si on s'arrête on répond « oui ». Cet algorithme termine toujours et décide le problème de l'arrêt, ce qui est impossible : c'est donc que $\mathscr{F}$ n'était pas décidable. ✓

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.

Corrigé. **(1)** Une configuration de l'exécution de $\mathscr{M}$ est la donnée d'un état parmi au plus m, d'une position de la tête parmi au plus $p(n)$ (puisque la tête visite au plus ce nombre de cellules), et de la valeur de chaque cellule du ruban ; or au plus $p(n) + n$ cellules peuvent contenir un 1 (à n'importe quel moment de l'exécution), car la tête n'en visite qu'au plus $p(n)$ et au plus n portent un 1 initialement : il y a donc au plus $2^{p(n)+n}$ configurations possibles du ruban, et au plus $m \times p(n) \times 2^{p(n)+n}$ configurations de l'ensemble de la machine.

(2) Si $\mathscr{M}$ retombe sur une configuration exacte qu'elle a déjà exécutée, elle exécutera de nouveau exactement les mêmes instructions et ne retombera indéfiniment sur cette configuration sans jamais finir. Comme on a supposé que le calcul terminait, c'est qu'il doit parcourir des configurations toutes distinctes, donc termine en au plus $m \times p(n) \times 2^{p(n)+n}$ étapes.

(3) On vient de montrer que le calcul de f par $\mathscr{M}$ fait en temps au plus $m \times p(n) \times 2^{p(n)+n}$. Mais ceci est une fonction p.r. de n (car p l'est, et que $k \mapsto 2^k$ l'est, et que m est une constante ici). Donc f est calculée en complexité en temps p.r., donc elle est elle-même p.r. ✓

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 $\tau\colon \mathbb{N} \to \mathbb{N}$ 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.

Corrigé. **(1)** La fonction de lecture $(\tau, i) \mapsto \tau(i)$ consiste à parcourir tous les couples de la liste $\langle\langle\langle i_1, \tau(i_1)\rangle\rangle, \ldots, \langle i_k, \tau(i_k)\rangle\rangle\rangle$ qui représente le tableau et, pour chacune, comparer i à la première composante et, s'il y a égalité, renvoyer la seconde composante, sinon renvoyer 0. La boucle est bornée a priori car elle parcourt une liste connue (dont la longueur est majorée par le numéro qui la code). Comme le décodage des couples (et donc des listes) est primitif récursif, tout ceci est primitif récursif.

La fonction d'écriture $(\tau, i, v) \mapsto \tau'$ consiste à parcourir la liste qui représente le tableau, et si on a $i = i_r$, modifier la seconde composante du couple correspondant, tandis que si on a $i_r < i < i_{r+1}$ (ou $i < i_0$ ou $i > i_k$) on insère un nouveau couple : comme l'encodage et le décodage des couples (et notamment l'insertion d'un élément dans une liste) sont primitifs récursifs, tout ceci est primitif récursif.

(2) Le tableau est codé sous forme d'entier naturel comme on l'a dit, donc il devient une simple variable de boucle, sur laquelle on peut effectuer des lectures et modifications par les fonctions qu'on a expliquées (et qui sont primitives récursives). Le fait de disposer d'une variable dans une boucle (bornée !) pour un algorithme primitif récursif est bien permis (essentiellement par la récursion primitive, qui permet précisément la modification d'une variable à chaque tour de boucle). ✓

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

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

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

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 $B\colon \mathbb{N}^4 \to \mathbb{N}$:

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

(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.)

Corrigé. **(1)** On commence par remarquer que les « petites valeurs » $m \leq 1$ ou $n \leq 1$ de la fonction d'Ackermann, se calculent facilement par des formules de l'énoncé.

L'algorithme calculant $A(m,n,k)$ est alors le suivant. Si $m \leq 1$ ou $n \leq 1$ on peut facilement calculer la valeur comme on vient de l'expliquer, donc on se place dans le cas $m \geq 2$ et $n \geq 2$. (Notons aussi que toute valeur de la fonction d'Ackermann pour $m \geq 1$ est non nulle, ce qui nous permet d'utiliser 0 pour représenter « non calculé » dans le tableau.)

On initialise un tableau τ de deux indices, i, ℓ, initialement rempli de 0, qui servira à stocker les valeurs de la fonction d'Ackermann $A(m,i,\ell)$. Pour chaque ℓ allant de 0 à k et chaque i allant de 0 à b, on calcule $A(m,i,\ell)$ de la manière suivante : si $i \leq 1$ ou $\ell = 0$ on utilise la formule évoquée ci-dessus et stocke la valeur dans le tableau ; sinon, on consulte le tableau en $(i-1,\ell)$ et, si cette valeur u est définie (c'est-à-dire non nulle), on consulte le tableau en $(u,\ell-1)$ et, si cette valeur w est définie, on la stocke dans le tableau en (i,ℓ).

Autrement dit :

- si $m \leq 1$ ou $n \leq 1$, calculer facilement $A(m,n,k)$ et renvoyer sa valeur ; sinon :
- initialiser un tableau τ (d'indices i allant de 0 à b et ℓ allant de 0 à k, et initialement rempli de 0),
- pour ℓ allant de 0 à k,
  - pour i allant de 0 à b,
    - si $i \leq 1$ ou $\ell = 0$, calculer facilement $w := A(m,i,\ell)$ et stocker $\tau(i,\ell) \leftarrow w$,
    - sinon, consulter $u := \tau(i-1,\ell)$,
    - si $u \neq 0$, consulter $w := \tau(u,\ell-1)$,
    - si $w \neq 0$, stocker $\tau(i,\ell) \leftarrow w$.
- finalement : renvoyer $\tau(k,n)$ si elle est > 0, sinon « échec ».

En Python, avec de petites variations :

```
def ackermann_small(m,n,k):
  # Returns A(m,n,k) value if m<=1 or n<=1 or k<=2
  if k==0: return m+n
  if k==1: return m*n
  if k==2: return m**n
  if n==0: return 1
  if n==1: return m
  if m==0: return (n+1)%2
  if m==1: return 1

def ackermann_bounded(m,n,k,b):
  # Returns A(m,n,k) (at least) if its value is <=b
  if m<=1 or n<=1: return ackermann_small(m,n,k)
  tab = {}
  for l in range(k+1):
    for i in range(b+1):
      if l==0 or i<=1:
        w = ackermann_small(m,i,l)
        tab[(i,l)] = w
      else:
        if (i-1,l) in tab:
          u = tab[(i-1,l)]
          if (u,l-1) in tab:
            w = tab[(u,l-1)]
            tab[(i,l)] = w
  if (n,k) in tab:
```

```
return tab[(n,k)]
```

L'algorithme repose sur la formule $A(m,i,\ell) = A(m, A(m,i-1,\ell), \ell-1)$ (qui est une simple réécriture de la troisième ligne de la définition), où on a appelé $u = A(m,i-1,\ell)$ et $w = A(m,u,\ell-1)$: cete formule montre que si les valeurs $(i-1,\ell)$ et $(u,\ell-1)$ sont trouvées dans le tableau, la valeur $A(m,i,\ell)$ sera correctement calculée.

Or on montre par récurrence sur ℓ et i que toutes les valeurs pour lesquelles $A(m,i,\ell) \leq b$ (ou bien $i \leq 1$ ou $\ell = 0$) seront effectivement calculées par l'algorithme : en effet, si $A(m,i,\ell) \leq b$, alors par la croissance en la deuxième variable de la fonction d'Ackermann, $u := A(m,i-1,\ell)$ est lui-même $\leq b$ (ou alors $i = 1$), donc aura été correctement calculé avant $A(m,i,\ell)$, et $A(m,u,\ell-1)$ aura été calculé et stocké dans le tableau puisque la boucle sur ℓ est extérieure à celle sur i et que la valeur u est dans les bornes de la boucle sur i.

En particulier, si b est un majorant de la valeur $A(m,n,k)$ qu'on cherchait, alors l'algorithme renvoie $A(m,n,k)$.

(2) L'algorithme qu'on a décrit ci-dessus ne fait aucun appel récursif et n'utilise que des boucles bornées (deux boucles imbriquées). L'utilisation d'un tableau est justifiée par l'exercice 1.13. On a donc bien défini une fonction primitive récursive.

(3) L'algorithme qu'on a décrit calcule de façon primitive récursive en m,n,k,b la valeur $A(m,n,k)$ si tant est que celle-ci est $\leq b$. En particulier, pour calculer $B(m,n,k,v)$ il suffit d'appliquer cet algorithme à m,n,k,v (c'est-à-dire avec v lui-même comme borne) et, s'il renvoie une valeur w, tester si $v = w$: si c'est le cas on renvoie « oui » (enfin, 1), sinon, ou si l'algorithme n'a pas réussi à caluler $A(m,n,k)$ on renvoie « non » (enfin, 0). La fonction B est donc primitive récursive.

(On dit parfois abusivement que la fonction d'Ackermann a un graphe primitif récursif pour dire que la fonction indicatrice de son graphe est primitive récursive. On comparera à l'exercice 1.7 d'après lequel une fonction dont la fonction indicatrice du graphe est calculable, i.e., générale récursive, est elle-même calculable, i.e., générale récursive.) ✓

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.

Corrigé. **(1)** Si E est décidable tel que $L \subseteq E$ et $M \subseteq \complement E$, alors un algorithme qui décide E (c'est-à-dire, quand on lui fournit l'entrée x, répond « vrai » si $x \in E$, et « faux » si $x \notin E$) répond bien aux critères demandés. Réciproquement, donné un algorithme qui répond aux critères demandés, si E est l'ensemble des x sur lesquels il répond « vrai », alors E est bien décidable (on peut toujours modifier l'algorithme si nécessaire pour qu'il ne réponde que « vrai » ou « faux »), et on a $L \subseteq E$ et $M \subseteq \complement E$.

(2) Si L, M sont décidables disjoints, on peut poser $E = L$, qui est décidable et vérifie à la fois $L \subseteq E$ (trivialement) et $M \subseteq \complement E$ (c'est une reformulation du fait que M est disjoint de $E = L$).

(3) Comme L est l'ensemble des codes des couples $\langle e, x\rangle$ tels que $\varphi_e(x) = 1$ et M l'ensemble des codes des couples $\langle e, x\rangle$ tels que $\varphi_e(x) = 2$, aucun élément ne peut appartenir aux deux, c'est-à-dire qu'ils sont disjoints.

(4) Pour semi-décider si le code d'un couple $\langle e, x\rangle$ appartient à L, il suffit de lancer l'exécution du programme e sur l'entrée x et, si elle termine en retournant 1, renvoyer « vrai », tandis que si elle termine en renvoyant n'importe quelle autre valeur, faire une boucle infinie (bien sûr, si le programme e ne termine jamais sur l'entrée x, on ne termine pas non plus). Ceci montre que L est semi-décidable. Le même raisonnement s'applique pour M.

(5) Supposons par l'absurde qu'il existe un algorithme g comme annoncé (i.e., qui prend $\langle e, x\rangle$ en entrée, termine toujours, et renvoie « vrai » si $\langle e, x\rangle \in L$ et « faux » si $\langle e, x\rangle \in M$). Définissons un nouvel algorithme qui, donné un entier e, effectue les calculs suivants : (1º) interroger l'algorithme g supposé exister en lui fournissant le code du couple $\langle e, e\rangle$ comme entrée, et ensuite (2º) si g répond vrai, renvoyer la valeur 2, tandis que si g répond n'importe quoi d'autre, renvoyer la valeur 1. L'algorithme qui vient d'être décrit aurait un certain numéro, disons, c, et la description de l'algorithme fait qu'il termine toujours, que la valeur $\varphi_c(e)$ qu'il renvoie vaut toujours soit 1 soit 2, et qu'elle vaut 2 si $\langle e, e\rangle \in L$ (c'est-à-dire si $\varphi_e(e) = 1$) et 1 si $\langle e, e\rangle \in M$ (c'est-à-dire si $\varphi_e(e) = 2$). En particulier, en prenant $e = c$, on voit que $\varphi_c(c)$ doit valoir 1 ou 2, doit valoir 2 si $\varphi_c(c) = 1$ et 1 si $\varphi_c(c) = 2$, ce qui est une contradiction.

Variante : La preuve ci-dessus a été rédigée en explicitant l'argument diagonal. On peut aussi, si on préfère, utiliser le théorème de récursion de Kleene. L'argument est alors le suivant. Supposons par l'absurde qu'il existe un algorithme g comme annoncé (i.e., qui prend $\langle e, x\rangle$ en entrée, termine toujours, et renvoie « vrai » si $\langle e, x\rangle \in L$ et « faux » si $\langle e, x\rangle \in M$). Définissons un nouvel algorithme qui, donné un couple (e, x), effectue les calculs suivants : (1º) interroger l'algorithme g supposé exister en lui fournissant le code $\langle e, x\rangle$ comme entrée, et ensuite (2º) si g répond vrai, renvoyer la valeur 2, tandis que si g répond n'importe quoi d'autre, renvoyer la valeur 1. On obtient ainsi une fonction h calculable totale $\mathbb{N}^2 \to \{1, 2\}$ telle que $h(e, x) = 2$ lorsque $\langle e, x\rangle \in L$ et $h(e, x) = 1$ lorsque $\langle e, x\rangle \in M$. Le théorème de récursion de Kleene assure qu'il existe e tel que $\varphi_e(x) = h(e, x)$ pour tout x, et notamment, quelle que soit x la valeur $\varphi_e(x)$ et définie et vaut soit 1 soit 2, et elle vaut 2 si $\langle e, x\rangle \in L$ (c'est-à-dire si $\varphi_e(x) = 1$) et 1 si $\langle e, x\rangle \in M$ (c'est-à-dire si $\varphi_e(x) = 2$). Ceci est une contradiction.

(6) La question (5) montre (compte tenu de la question (1)) que L et M ne sont pas calculablement séparables, i.e.., sont calculablement inséparables, tandis que (3) et (4) montrent que L et M sont disjoints et semi-décidables. On a donc bien montré l'existence d'ensembles semi-décidables disjoints et calculablement inséparables. ✓

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.)

Corrigé. **(1)** On construit le programme q suivant : il invoque le programme p sur q lui-même et ensuite, si p termine et renvoie X ou Y, il échange X et Y (autrement dit, si p appelé sur q renvoie X, alors il renvoie Y, et si p appelé sur q renvoie Y, alors il renvoie X), tandis que dans tout autre cas il fait une boucle infinie. La construction de ce q, et notamment le fait que q fasse appel à lui-même dans sa définition, est justifiée par l'astuce de Quine (formellement : si

on note $\varphi_p(q)$ le résultat du programme p invoqué sur q, on appelle h la fonction qui à q associe X si $\varphi_p(q) = Y$ et Y si $\varphi_p(q) = X$ et $\uparrow$ dans tout autre cas, alors h est calculable, donc par le théorème de récursion de Kleene, il existe q tel que le résultat $\varphi_q(0)$ de q soit $h(q)$).

Alors le résultat de p invoqué sur q ne peut pas être le mauvais choix, car si c'était le mauvais choix, q tout seul renverrait le bon choix (par construction de q), donc p invoqué sur q renverrait le bon choix (par la garantie fournie sur p), ce qui est une contradiction.

(2) Considérons le programme p qui prend en entrée un programme q, l'exécute (sans argument) et renvoie son résultat. Cette opération est bien algorithmique d'après l'existence d'une machine universelle. Or le programme ainsi défini répond à la garantie exigée de p : si q termine en renvoyant le bon choix, alors p appelé avec q en argument termine aussi en renvoyant le bon choix. Comme ce programme ne dépend pas de l'identité du bon choix (qui peut encore être X ou Y), il ne permet pas de trancher entre les deux : il est donc impossible de s'en servir pour faire un programme qui termine à coup sûr et renvoie le bon choix. (Plus exactement, si on avait un tel moyen de faire, ce moyen devrait s'appliquer encore quand on échange X et Y, ce qui contredit le fait qu'il renvoie toujours 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 $\varphi_{p'}$?

Corrigé. **(1)** Le programme p fonctionne ainsi : il teste si le programme q qu'on lui a passé en entrée est p lui-même (cette autoréférence est permise par l'astuce de Quine) et, si oui, renvoie 42, sinon, il invoque la fonction universelle $(e, i) \mapsto \varphi_e(i)$ pour calculer $\varphi_q(q)$.

De façon plus soigneuse et formelle : soit $h(p, q)$ valant 42 si $q = p$ et $\varphi_q(q)$ sinon ; cette fonction partielle $h \colon \mathbb{N}^2 \dashrightarrow \mathbb{N}$ est manifestement récursive (étant calculée par l'algorithme : « comparer les deux arguments, renvoyer 42 s'ils sont égaux, et sinon invoquer la fonction universelle sur le second argument deux fois »). Par le théorème de récursion de Kleene, il existe p tel que $\varphi_p(q) = h(p, q)$: on a donc bien $\varphi_p(p) = 42$ et $\varphi_p(q) = \varphi_q(q)$ pour tout autre q (mais pour $q = p$ cette relation est triviale donc elle vaut encore).

(2) Le programme p exécuté sur p' renvoie $\varphi_p(p') = \varphi_{p'}(p') = 1729$ (la première égalité découlant de la définition de p et la seconde de celle de p'). Symétriquement, le programme p' exécuté sur p renvoie $\varphi_{p'}(p) = \varphi_p(p) = 42$ (la première égalité découlant de la définition de p' et la seconde de celle de p).

Les fonctions partielles $e \mapsto \varphi_p(e)$ et $e \mapsto \varphi_{p'}(e)$ sont *égales* puisque toutes les deux coïncident avec $e \mapsto \varphi_e(e)$ en tout point d'après la définition de p et p'.

Remarque : Ce qui rend cet exercice légèrement « paradoxal », c'est qu'on a construit deux programmes p, p' qui calculent la *même* fonction (comme on vient de l'expliquer), et pourtant, p appelé sur lui-même renvoie 42 tandis que p' appelé sur lui-même renvoie 1729. Ce n'est pourtant pas si mystérieux quand on se rappelle que l'intention d'un programme (son code) n'est pas la même chose que l'extension (ce qu'il calcule). ✓

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.

Corrigé. *Première approche possible :* En se rappelant la manière dont on a numéroté les fonctions générales récursives, $e' = \langle\!\langle 3, 1, d, e \rangle\!\rangle$ convient, où d est un code de h (c'est-à-dire $h = \varphi_d$), or la fonction $e \mapsto \langle\!\langle 3, 1, d, e \rangle\!\rangle$ est primitive récursive.

Deuxième approche possible : On considère l'algorithme qui, donné e et n, calcule d'abord $\varphi_e(n)$ (ceci est faisable grâce à l'existence d'un interpréteur universel), puis lui applique h (ceci est faisable car h est calculable), et renvoie le résultat. Ceci calcule la fonction $(e, n) \mapsto h(\varphi_e(n))$, qui est donc calculable : appelons p un code de celle-ci comme fonction générale récursive. Par le théorème s-m-n (et en notant comme dans le cours $s_{1,1}$ la fonction de substitution d'une variable dans un programme de deux variables) on a $\varphi_{s_{1,1}(p,e)}(n) = \varphi_p(e, n) = h(\varphi_e(n)) = (h \circ \varphi_e)(n)$, donc la fonction $H \colon n \mapsto s_{1,1}(p, e)$ convient, et elle est bien primitive récursive.

Remarque : La première approche a l'avantage de fonctionner encore pour les fonctions primitives récursives (i.e., si h est supposée primitive récursive et qu'on remplace φ_e par la numérotation correspondante ψ_e des fonctions primitives récursives. Elle a l'inconvénient de dépendre des détails du codage des fonctions générales récursives. La seconde approche montre que cette dépendance est illusoire. ✓

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)** $(\lambda xy.xy)(\lambda xy.xy)$

Corrigé. **(a)** $(\lambda x.x)(\lambda x.x) \to_\beta \lambda x.x$ **(b)** $(\lambda x.xx)(\lambda x.x) \to_\beta (\lambda x.x)(\lambda x.x) \to_\beta \lambda x.x$ **(c)** $(\lambda x.xx)(\lambda x.xx) \to_\beta (\lambda x.xx)(\lambda x.xx) \to_\beta \cdots$ la seule β-réduction possible boucle donc il n'y a pas de forme normale. **(d)** On renomme d'abord les variables liées en se rappelant que chaque variable est liée par le λ le plus *intérieur* sur son nom : $(\lambda xx.x)(\lambda xx.x) = (\lambda x.\lambda x.x)(\lambda x.\lambda x.x) \equiv_\alpha (\lambda x.\lambda y.y)(\lambda u.\lambda v.v) \to_\beta \lambda y.y \equiv_\alpha \lambda x.x$ **(e)** $(\lambda xy.x)(\lambda xy.x) = (\lambda x.\lambda y.x)(\lambda x.\lambda y.x) \to_\beta \lambda y.\lambda x.\lambda y.x \equiv_\alpha \lambda y.\lambda x.\lambda z.x = \lambda yxz.x$ **(f)** $(\lambda xy.xy)y = (\lambda x.\lambda y.xy)y$ ici pour faire la β-réduction on doit d'abord renommer la variable liée par le second λ pour éviter qu'elle capture le y libre : $(\lambda x.\lambda y.xy)y \equiv_\alpha (\lambda x.\lambda z.xz)y \to_\beta \lambda z.yz$ (le piège serait de répondre $\lambda y.yy$ ici !) **(g)** $(\lambda xy.xy)(\lambda xy.xy) = (\lambda x.\lambda y.xy)(\lambda x.\lambda y.xy) \to_\beta \lambda y.(\lambda x.\lambda y.xy)y \equiv_\alpha \lambda y.(\lambda x.\lambda z.xz)y \to_\beta \lambda y.\lambda z.yz = \lambda yz.yz \equiv_\alpha \lambda xy.xy$ ✓

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.

Corrigé. **(1)** La β-réduction du seul redex de T_2 s'écrit $(\lambda x.xxx)(\lambda x.xxx) \to (\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx)$. Appelons T_3 le terme en question, et plus généralement T_n le terme $(\lambda x.xxx)\cdots(\lambda x.xxx)$ avec $n-1$ applications sur $(\lambda x.xxx)$ de $(\lambda x.xxx)$ (donc n fois ce sous-terme au total) ; on se rappellera bien que les parenthèses sont vers la *gauche*, c'est-à-dire que T_4 est $((T_1T_1)T_1)T_1$ par exemple. Il y a un *unique* redex dans T_n (bien qu'il y ait n lambdas, un seul est appliqué), à savoir celui des deux T_1 les plus à gauche (ou les plus profondément imbriqués) : la seule β-réduction possible consiste à remplacer ce redex T_1T_1 (soit T_2) par son réduit $(T_1T_1)T_1$ (soit T_3), ce qui donne T_{n+1}. Le graphe des β-réductions est donc $T_2 \to T_3 \to T_4 \to \cdots$ avec une unique β-réduction possible à chaque fois. Le terme n'est pas faiblement (ni à plus forte raison fortement) normalisable.

(2) La β-réduction du seul redex donne $(\lambda x.x(xx))(\lambda x.x(xx)) \to (\lambda x.x(xx))((\lambda x.x(xx))(\lambda x.x(xx)))$. Notant $U := \lambda x.x(xx)$ pour y voir plus clair, on a donc $V := UU \to U(UU)$. Maintenant on a deux possibilités de redex à réduire : le redex extérieur $U(UU)$ formé par l'expression tout entière, et le redex intérieur UU. Le redex extérieur $U(UU)$ se réduit en $(UU)((UU)(UU))$ (qui a maintenant trois redex), tandis que la réduction du redex intérieur UU dans $U(UU)$ donne $U(U(UU))$. Il est alors facile de construire toutes sortes de chemins de β-réductions en se rappelant que tout UX est un redex, avec pour réduit $X(XX)$. On peut notamment distinguer la réduction intérieure gauche (ou droite, ici elles coïncident)

$$UU \to U(UU) \to U(U(UU)) \to U(U(U(UU))) \to U(U(U(U(UU)))) \cdots$$

et la réduction extérieure gauche (on utilise $V := UU$ pour plus de clarté ; mais on gardera bien à l'esprit que V, contrairement à U, n'est pas une abstraction donc ne forme pas un redex quand on l'applique, par contre c'est lui-même un redex qui se réduit en UV)

$$\begin{aligned} UU := V &\to UV \to V(VV) \to (UV)(VV) \to (V(VV))(VV) \\ &\to ((UV)(VV))(VV) \to ((VV)(VV))(VV) \to \cdots \end{aligned}$$

ou encore la réduction extérieure droite

$$\begin{aligned} UU := V \to UV \to V(VV) \to V(V(UV)) \to V(V(V(VV))) \\ \to V(V(V(UV)))) \to V(V(V(V(VV))))) \to \cdots \end{aligned}$$

Aucun de ces chemins ne termine (on a vu en cours que si la réduction extérieure gauche ne termine pas, aucun chemin de β-réductions ne termine, mais ici c'est clair car une β-réduction ne peut de toute façon qu'augmenter le nombre de U dans l'expression).

(Seule la notion de réduction extérieure gauche a été définie en cours ; on peut néanmoins définir sans difficulté les quatre réductions extérieure gauche, intérieure gauche, extérieure droite et intérieure droite : le redex extérieur gauche est celui dont le bord gauche est le plus à gauche, le redex intérieur gauche est celui dont le bord droit est le plus à gauche, le redex extérieur droite est celui dont le bord droit est le plus à droite, et le redex intérieur droite est celui dont le bord gauche est le plus à droite. Peu importent ces définitions, cependant, ici le but est simplement d'illustrer quelques chemins possibles.)

(*Remarque :* Je n'ai pas réfléchi à trouver une caractérisation de tous les termes en lesquels UU peut se réduire, mais on peut les décrire comme des arbres d'application avec U aux feuilles, et la β-réduction se voir alors comme une transformation simple sur les arbres.)

(3) La β-réduction du seul redex de R s'écrit $(\lambda x.\lambda v.xxv)(\lambda x.\lambda v.xxv) \to \lambda v.(\lambda x.\lambda v.xxv)(\lambda x.\lambda v.xxv)v = \lambda v.Rv$. On peut alors continuer ainsi : $R \to \lambda v.Rv \to \lambda v.(\lambda v.Rv)v \to \lambda v.(\lambda v.(\lambda v.Rv)v)v \to \cdots$. Même si ces écritures sont correctes (rappelons que chaque variable est liée par le λ le plus *intérieur* sur son nom), il est considérablement plus clair de renommer les variables liées, par exemple ainsi :

$$R \to \lambda v_1.Rv_1 \to \lambda v_1.(\lambda v_2.Rv_2)v_1 \to \lambda v_1.(\lambda v_2.(\lambda v_3.Rv_3)v_2)v_1 \to \cdots$$

(on prendra garde à ne pas confondre le troisième terme de cette suite, par exemple, avec $\lambda v_1.\lambda v_2.Rv_2v_1$ qui désigne $\lambda v_1.\lambda v_2.(Rv_2v_1)$ et qui peut s'écrire $\lambda v_1v_2.Rv_2v_1$: ce n'est pas du tout la même chose !). ✓

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.

Corrigé. **(a)** En Python : $\overline{0}$ devient `lambda f: lambda x: x`, $\overline{1}$ devient `lambda f: lambda x: f(x)`, $\overline{2}$ devient `lambda f: lambda x: f(f(x))` (chacun sur une ligne) et $\overline{3}$ devient `lambda f: lambda x: f(f(f(x)))` (chacun sur une ligne). En Scheme : $\overline{0}$ devient `(lambda (f) (lambda (x) x))`, $\overline{1}$ devient `(lambda (f) (lambda (x) (f x)))`, $\overline{2}$ devient `(lambda (f) (lambda (x) (f (f x))))` et $\overline{3}$ devient `(lambda (f) (lambda (x) (f (f (f x)))))` (espacement indifférent mais les parenthèses sont critiques).

(b) Pour convertir un entier de Church en entier natif, il suffit d'itérer la fonction successeur la nombre de fois représenté par l'entier de Church, ce que l'entier de Church permet justement de faire, en l'appliquant au final à 0. En Python, cela donne : `def fromchurch(ch): return (ch (lambda n: n+1))(0)` (ou `fromchurch = lambda ch: (ch (lambda n: n+1))(0)` mais dans tous les cas sur une seule ligne) ; en Scheme : `(define (fromchurch ch) ((ch (lambda (n) (+ n 1))) 0))` ce qui est du sucre syntaxique pour `(define fromchurch (lambda (ch) ((ch (lambda (n) (+ n 1))) 0)))`.

(c) Voici un exemple de code vérifiant que $2+3=5$, que $2\times 3=6$ et que $2^3=8$ sur les entiers de Church, d'abord en Python :

```
churchzero = (lambda f: lambda x: x)
churchone = (lambda f: lambda x: f(x))
churchtwo = (lambda f: lambda x: f(f(x)))
churchthree = (lambda f: lambda x: f(f(f(x))))
fromchurch = lambda ch: (ch (lambda n: n+1))(0)
churchadd = lambda m: lambda n: lambda f: lambda x: (n(f))((m(f))(x))
churchmul = lambda m: lambda n: lambda f: n(m(f))
churchpow = lambda m: lambda n: n(m)
# Check 2+3 == 5:
fromchurch((churchadd(churchtwo))(churchthree))
# Check 2*3 == 6:
fromchurch((churchmul(churchtwo))(churchthree))
# Check 2^3 == 8:
fromchurch((churchpow(churchtwo))(churchthree))
```

…puis en Scheme :

```
(define churchzero (lambda (f) (lambda (x) x)))
(define churchone (lambda (f) (lambda (x) (f x))))
(define churchtwo (lambda (f) (lambda (x) (f (f x)))))
(define churchthree (lambda (f) (lambda (x) (f (f (f x))))))
(define fromchurch (lambda (ch) ((ch (lambda (n) (+ n 1))) 0)))
(define churchadd (lambda (m) (lambda (n) (lambda (f) (lambda (x)
((n f) ((m f) x)))))))
(define churchmul (lambda (m) (lambda (n) (lambda (f) (n (m f))))))
(define churchpow (lambda (m) (lambda (n) (n m))))
;; Check 2+3 == 5:
(fromchurch ((churchadd churchtwo) churchthree))
;; Check 2*3 == 6:
(fromchurch ((churchmul churchtwo) churchthree))
;; Check 2^3 == 8:
(fromchurch ((churchpow churchtwo) churchthree))
```

Dans les deux cas, les valeurs retournées sont successivement 5, 6 et 8.

Noter que dans les deux langages la syntaxe est rendue lourdingue par le fait que (conformément aux conventions du λ-calcul dont on a mécaniquement traduit des termes) on ne crée que des fonctions d'*un* argument, ce qui oblige les fonctions d'opération à prendre les arguments sous forme « curryfiée ». Il serait bien plus naturel d'écrire par exemple `churchpow = lambda m,n: n(m)` en Python et `(define churchpow (lambda (m n) (n m)))` en Scheme pour définir directement une fonction de *deux* arguments, qu'on peut ensuite utiliser comme `churchpow(churchtwo,churchthree)` et `(churchpow churchtwo churchthree)` respectivement.

`(d)` En Python : `(lambda x: x(x))(lambda x: x(x))` ; en Scheme : `((lambda (x) (x x)) (lambda (x) (x x)))`. Le premier termine rapidement avec un débordement de pile (au moins dans la version actuelle Python 3.11), le second, quel que soit l'interpréteur Scheme (au moins tous ceux que j'ai pu tester), boucle indéfiniment (mais sans consommation de pile supplémentaire ni d'autre forme de mémoire).

La raison de cette différence est que Scheme effectue (et la spécification du langage impose) une *récursion terminale*

propre : lorsque le code d'une fonction f termine par l'appel à une autre fonction g (en renvoyant sa valeur), le contrôle de l'exécution est simplement passé de f à g sans empilement d'adresse de retour (qui n'a pas lieu d'être puisque la valeur de retour de f sera justement celle de g) ; en Python, en revanche, la récursion terminale n'est pas traitée spécialement, donc chaque appel à `x(x)` est empilé et jamais dépilé et la pile déborde rapidement.

`(e)` La traduction du λ-calcul en OCaml ou Haskell est évidente en utilisant `fun v -> E` (noté `\v -> E` en Haskell) pour traduire $\lambda v.E$ (et toujours `(P Q)` pour (PQ)). Néanmoins, il n'est pas évident qu'on puisse toujours écrire les termes qu'on souhaite, parce qu'ils ne seront pas forcément typables.

En OCaml, le test sur les entiers de Church donne :

```
let churchzero = fun f -> fun x -> x
let churchone = fun f -> fun x -> f x
let churchtwo = fun f -> fun x -> f (f x)
let churchthree = fun f -> fun x -> f (f (f x))
let fromchurch = fun ch -> ch (fun n->(n+1)) 0
let churchadd = fun m -> fun n -> fun f -> fun x -> (n f)(m f x)
let churchmul = fun m -> fun n -> fun f -> n (m f)
let churchpow = fun m -> fun n -> n m
;;
(* Check 2+3 == 5: *)
fromchurch(churchadd churchtwo churchthree) ;;
(* Check 2*3 == 6: *)
fromchurch(churchmul churchtwo churchthree) ;;
(* Check 2^3 == 8: *)
fromchurch(churchpow churchtwo churchthree) ;;
```

…et en Haskell :

```
let churchzero = \f -> \x -> x
let churchone = \f -> \x -> f x
let churchtwo = \f -> \x -> f (f x)
let churchthree = \f -> \x -> f (f (f x))
let fromchurch = \ch -> ch (\n->(n+1)) 0
let churchadd = \m -> \n -> \f -> \x -> (n f)(m f x)
let churchmul = \m -> \n -> \f -> n (m f)
let churchpow = \m -> \n -> n m
- Check 2+3 == 5:
fromchurch(churchadd churchtwo churchthree)
- Check 2*3 == 6:
fromchurch(churchmul churchtwo churchthree)
- Check 2^3 == 8:
fromchurch(churchpow churchtwo churchthree)
```

Il se trouve que sur ces exemples simples le typage n'empêche pas la construction, mais si on essayait de faire la fonction $n \mapsto n^n$, par exemple, la fonction `fun n -> churchpow n n` ne type pas. De même, le terme non normalisant de la question (d), qui se traduirait `(fun x -> x x)(fun x -> x x)` en OCaml, et `(\x -> x x)(\x -> x x)` en Haskell, est refusé par le système de typage : ni le OCaml ni le Haskell ne permet de traduire tous les termes du λ-calcul non typé, précisément parce qu'ils sont typés.

(**Attention :** si le Python comme le Scheme permettent de *traduire* tous les termes du λ-calcul non typé, le comportement de l'évaluateur, dans les deux cas, ne correspond pas forcément à une stratégie évidente de β-réduction du λ-calcul. Notamment, le terme $(\lambda uz.z)((\lambda x.xx)(\lambda x.xx))$, bien que faiblement normalisable en λ-calcul, conduira une fois traduit à une boucle dans ces deux langages, parce que l'évaluateur commence par évaluer les arguments d'une fonction avant d'appliquer la fonction ; inversement, le terme $\lambda u.(\lambda x.xx)(\lambda x.xx)$, bien qu'il ne soit même pas faiblement normalisable en λ-calcul, est accepté sans broncher par ces deux langages car le corps d'une fonction n'est évalué qu'à l'application de la fonction. Cependant, les calculs de fonctions primitives récursives sur les entiers de Church ne font intervenir que des termes fortement normalisants sur lesquels ces difficultés ne se posent pas.) ✓

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 ?

Corrigé. **(1)** Effectuons par exemple la β-réduction extérieure gauche (mais on rappelle que le théorème de Church-Rosser affirme que la normalisation est confluente : tout chemin de β-réduction peut rejoindre tout autre chemin, notamment si on arrive à une forme normale ce sera la même) : $\pi_1(\Pi XY) = (\lambda p.p(\lambda xy.x))((\lambda xyf.fxy)XY) \to ((\lambda xyf.fxy)XY)(\lambda xy.x) \to\to (\lambda f.fXY)(\lambda xy.x) \to (\lambda xy.x)XY \to\to X$. Le résultat est le même, *mutatis mutandis*, pour π_2, à savoir : $\pi_2(\Pi XY) = (\lambda p.p(\lambda xy.y))((\lambda xyf.fxy)XY) \to ((\lambda xyf.fxy)XY)(\lambda xy.y) \to\to (\lambda f.fXY)(\lambda xy.y) \to (\lambda xy.y)XY \to\to Y$.

(2) Le couple (x, y) est codé par Π en le terme Πxy c'est-à-dire (à β-réduction près) la fonction $\lambda f.fxy$ qui prend une fonction f et l'applique (de façon « currifiée ») aux deux composantes du couple. (Autrement dit, pour appliquer une fonction au couple, on applique la représentation du couple à la fonction !) Pour décoder le couple, il s'agit simplement d'utiliser pour f la fonction $\lambda xy.x$ qui renvoie son premier argument lorsqu'on veut récupérer celui-ci, et c'est ce que fait π_1, ou la fonction $\lambda xy.x$ qui renvoie son premier argument lorsqu'on veut récupérer celui-ci, et c'est ce que fait π_2.

(3) Voici une implémentation en Scheme, dans laquelle on a pris la liberté d'utiliser des fonctions de plusieurs variables (le Scheme permet de définir des fonctions de plusieurs variables sans passer les arguments un par un de façon « curryfiée » : la notation est `(f x y)` pour appeler une telle fonction `f` sur deux arguments `x` et `y`, et `(lambda (x y) ...)` pour en définir une ; par ailleurs, les fonction `cons`, `car` et `cdr` du Scheme sont les fonctions servant nativement à créer et projeter des paires, i.e., ce sont les équivalents natifs des fonctions `pairing`, `proj1` et `proj2` qu'on définit ici) :

```
(define pairing (lambda (x y) (lambda (f) (f x y))))
(define proj1 (lambda (p) (p (lambda (x y) x))))
(define proj2 (lambda (p) (p (lambda (x y) y))))
(define fromnative (lambda (z) (pairing (car z) (cdr z))))
(define tonative (lambda (p) (p cons)))
```

On peut ensuite faire différents tests, par exemple `(proj1 (pairing 42 "coucou"))` renvoie `42`, comme `(proj1 (fromnative (cons 42 "coucou")))` ; et `(tonative (pairing 42 "coucou"))` renvoie la paire native, notée `(42 . "coucou")` en Scheme.

Voici maintenant le code équivalent en OCaml : il n'était pas évident *a priori* que le codage puisse être implémenté dans ce langage (i.e., qu'il soit typable), mais il s'avère qu'il l'est, modulo la subtilité qui sera expiquée ci-dessous :

```
let pairing = fun x -> fun y -> fun f -> f x y
let proj1 = fun p -> p (fun x -> fun y -> x)
let proj2 = fun p -> p (fun x -> fun y -> y)
(* Conversion from and to native pairs *)
let fromnative = fun (x,y) -> pairing x y
let tonative = fun p -> p (fun x -> fun y -> (x,y))
;;
```

On peut alors tester que `proj1 (pairing 42 "coucou")` renvoie `42`, comme `proj1 (fromnative (42,"coucou"))` ; et `tonative (pairing 42 "coucou")` renvoie la paire native, notée `(42, "coucou")` en OCaml.

Subtilité : Même si cette version OCaml fonctionne bien, une petite variation apparemment anodine, à savoir écrire `let tonative = fun p -> (proj1 p, proj2 p)` (notons que l'équivalent Scheme, `(define tonative (lambda (p) (cons (proj1 p) (proj2 p))))`, fonctionne parfaitement) pose des problèmes de typage : avec cette nouvelle définition, `tonative (pairing 42 1729)` fonctionne toujours, mais `tonative (pairing`

`42 "coucou")` conduit à une erreur de typage. Sans entrer dans les détails de cette erreur, le problème est que comme le type du résultat de la fonction `pairing` appliquée à deux types a et b est $(\forall c)(a \to b \to c) \to c$ (par exemple, celui du couple `pairing 42 "coucou"` est $(\forall c)(\texttt{int} \to \texttt{string} \to c) \to c$), c'est-à-dire une fonction polymorphe acceptant $f : a \to b \to c$ pour n'importe quel type c et renvoyant ce même type c ; la fonction `tonative` « devrait » donc avoir pour type $(\forall a, b)((\forall c)((a \to b \to c) \to c)) \to a \times b$ (ce qui *n'est pas* la même chose que $(\forall a, b, c)((a \to b \to c) \to c) \to a \times b$), i.e., elle devrait recevoir une fonction polymorphe en argument ; mais le système de typage de OCaml, basé sur l'algorithme de Hindley-Milner, ne peut exprimer que des fonctions polymorphes, pas des fonctions attendant une fonction polymorphe en argument, si bien que OCaml ne peut pas typer correctement la fonction `tonative` (et selon l'expression précise utilisée, on obtient des approximations plus ou moins bonnes du « vrai » type qu'on vient de dire). Voir l'exercice 5.1(2) pour un cadre donnant un sens précis aux types intervenant dans ce paragraphe.

Dans un quelconque de ces langages, si on implémente la fonction `pairing` comme on vient de le dire, la valeur de `x` et `y` dans `pairing x y` est stockée dans la *clôture* de la fonction renvoyée, c'est-à-dire les liaisons locales (de `x` et `y` à leurs valeurs respectives) qui ont été faites lors de la création de la fonction et qui peuvent, ainsi que le montre cet exemple, survivre bien au-delà de la portée de définition de la fonction dans un langage fonctionnel. ✓

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 $\neg\neg B$).

Corrigé. **(a)** On commence par se placer avec $f_1 : \neg\neg(A \Rightarrow B)$ et $x_1 : \neg\neg A$ et $k : \neg B$ dans le contexte. Appelons encore f_0 l'hypothèse $A \Rightarrow B$. Le raisonnement « si on a A, on a B, ce qui contredit $\neg B$ » se formalise par le λ-terme $\lambda(x_0 : A).k(f_0 x_0)$ de type $\neg A$; le « mais ceci contredit $\neg\neg A$ » s'obtient en lui appliquant x_1. Le λ-terme $\lambda(f_0 : A \Rightarrow B).\, x_1(\lambda(x_0 : A).k(f_0 x_0))$ est donc de type $\neg(A \Rightarrow B)$, et la contradiction avec $\neg\neg(A \Rightarrow B)$ s'exprime en lui appliquant f_1. Finalement, le λ-terme tout entier (de type $\neg\neg(A \Rightarrow B) \Rightarrow \neg\neg A \Rightarrow \neg\neg B$) est :

$$\lambda(f_1 : \neg\neg(A \Rightarrow B)).\, \lambda(x_1 : \neg\neg A).\, \lambda(k : \neg B).\, f_1(\lambda(f_0 : A \Rightarrow B).\, x_1(\lambda(x_0 : A).k(f_0 x_0)))$$

(ou en syntaxe Coq : `fun (f1 : ~~(A->B)) (x1 : ~~A) (k : ~B) => f1 (fun f0 : A -> B => x1 (fun x0 : A => k (f0 x0)))`).

(b) On commence par se placer avec $f_1 : A \Rightarrow \neg\neg B$ et $k : \neg(A \Rightarrow B)$ dans le contexte. Appelons y l'hypothèse B : alors $\lambda(z : A).y$ est de type $A \Rightarrow B$, donc $k(\lambda(z : A).y)$ est de type $\bot$, ainsi $\lambda(y : B).k(\lambda(z : A).y)$ est de type $\neg B$. Appelons x l'hypothèse A : alors $f_1 x$ montre $\neg\neg B$, donc en l'appliquant au λ-terme $\lambda(y : B).k(\lambda(z : A).y)$ précédemment trouvé on trouve une contradiction, et $\texttt{exfalso}^{(B)}(f_1 x\, \lambda(y : B).k(\lambda(z : A).y))$ est de type B. En abstrayant x dans ce terme on a un terme de type $A \Rightarrow B$, et en lui appliquant k on a la contradiction recherchée. Finalement, le λ-terme tout entier (de type $(A \Rightarrow \neg\neg B) \Rightarrow \neg\neg(A \Rightarrow B)$) est :

$$\lambda(f_1 : A \Rightarrow \neg\neg B).\, \lambda(k : \neg(A \Rightarrow B)).\, k(\lambda(x : A).\, \texttt{exfalso}^{(B)}(f_1 x\, \lambda(y : B).k(\lambda(z : A).y)))$$

(ou en syntaxe Coq : `fun (f1 : A -> ~~B) (k : ~(A->B)) => k (fun x : A => False_ind B (f1 x (fun y : B => k (fun z : A => y))))`).

(c) La preuve de $A \Rightarrow \neg\neg A$ s'écrit $\lambda(x : A).\lambda(k : \neg A).kx$. La preuve de $A \Rightarrow C$ à partir de $\neg\neg A \Rightarrow C$ s'obtient en composant avec cette fonction, donc finalement le λ-terme tout entier (de type $(\neg\neg A \Rightarrow \neg\neg B) \Rightarrow (A \Rightarrow \neg\neg B)$) est :

$$\lambda(f_1 : \neg\neg A \Rightarrow \neg\neg B).\, \lambda(x : A).f_1(\lambda(k : \neg A).kx)$$

(ou en syntaxe Coq : `fun (f1 : ~~A -> ~~B) (x : A) => f1 (fun k : ~A => k x)`).

Les trois raisonnements sont parfaitement valables intuitionnistement (malgré les nombreux « supposons (...), contradiction », il s'agit à chaque fois de *preuves de négation* et pas de *preuves par l'absurde*) : le fait qu'on ait trouvé des λ-termes (sans aucun call/cc dedans) de ces types le montre.

On peut retenir la conclusion que $\neg\neg(A \Rightarrow B)$, $\neg\neg A \Rightarrow \neg\neg B$ et $A \Rightarrow \neg\neg B$ sont tous les trois équivalents (en logique intuitionniste). ✓

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 $f\langle x, y\rangle$.

Corrigé. Posons $D := (A \Rightarrow C) \vee (B \Rightarrow C)$ pour abréger les notations.

La fonction qu'on va finalement renvoyer une fois capturé le x est $\iota_2^{(A\Rightarrow C, B\Rightarrow C)}(\lambda(y : B).\, f\langle x, y\rangle)$ (de type D) : on va donc invoquer la continuation (appelons-la k) sur cette valeur. La fonction provisoirement renvoyée est donc $\iota_1^{(A\Rightarrow C, B\Rightarrow C)}(\lambda(x : A).\, k(\cdots))$ où les points de suspension sont remplacés par la valeur qu'on vient de dire (ce terme est de type D et on voit que la continuation fait semblant de renvoyer un type C — sachant qu'en fait elle ne renverra jamais rien puisque c'est une continuation).

Il n'y a donc plus qu'à appliquer call/cc de type $((D \Rightarrow C) \Rightarrow D) \Rightarrow D$ à tout ça, ce qui donne :

$$\lambda(f : A \wedge B \Rightarrow C).\, \texttt{callcc}\ (\lambda(k : D \Rightarrow C).\, \iota_1^{(A\Rightarrow C, B\Rightarrow C)}(\lambda(x : A).\, k(\iota_2^{(A\Rightarrow C, B\Rightarrow C)}(\lambda(y : B).\, f\langle x, y\rangle)))))$$

terme de type $(A \wedge B \Rightarrow C) \Rightarrow D$ (où on rappelle $D = (A \Rightarrow C) \vee (B \Rightarrow C)$). ✓

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.

Corrigé. Elle est prouvable en logique classique comme on le voit en considérant son tableau de vérité : le seul cas où $A \Rightarrow B$ est faux est celui où A est vrai et B est faux, et le seul cas où $B \Rightarrow A$ est faux est celui où B est vrai et A est faux : on ne peut donc pas être dans ces deux cas à la fois. (Si on préfère, $A \Rightarrow B$ équivaut classiquement à $\neg A \vee B$, et $B \Rightarrow A$ équivaut classiquement à $A \vee \neg B$, donc la disjonction des deux équivaut classiquement à $\neg A \vee B \vee A \vee \neg B$, qui est classiquement vrai.)

Elle n'est pas prouvable en logique intuitionniste à cause de la propriété de la disjonction : si on avait $\vdash (A \Rightarrow B) \vee (B \Rightarrow A)$, on aurait $\vdash A \Rightarrow B$ ou bien $\vdash B \Rightarrow A$; mais ni $A \Rightarrow B$ ni $B \Rightarrow A$ seul n'est prouvable intuitionnistement car elles ne sont même pas prouvables classiquement (leur tableau de vérité n'est pas entièrement vrai). ✓

Exercice 3.4. ($\star$)

Montrer en calcul propositionnel intuitionniste que

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

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

(1)	$\boxed{A \vee B}$	
(2)	$\quad\boxed{A}$	
(3)	$\quad\boxed{A \Rightarrow B}$	
(4)	$\quad B$	$\Rightarrow$Élim sur (3) et (2)
(5)	$\quad(A \Rightarrow B) \Rightarrow B$	$\Rightarrow$Int de (3) dans (4)
(6)	$\quad\boxed{B \Rightarrow A}$	
(7)	$\quad(B \Rightarrow A) \Rightarrow A$	$\Rightarrow$Int de (6) dans (2)
(8)	$\quad((A \Rightarrow B) \Rightarrow B) \wedge ((B \Rightarrow A) \Rightarrow A)$	$\wedge$Int sur (5), (7)
(9)	$\quad\boxed{B}$	
(10)	$\quad\boxed{B \Rightarrow A}$	
(11)	$\quad A$	$\Rightarrow$Élim sur (10) et (9)
(12)	$\quad(B \Rightarrow A) \Rightarrow A$	$\Rightarrow$Int de (10) dans (11)
(13)	$\quad\boxed{A \Rightarrow B}$	
(14)	$\quad(A \Rightarrow B) \Rightarrow B$	$\Rightarrow$Int de (13) dans (9)
(15)	$\quad(((A \Rightarrow B) \Rightarrow B) \wedge (B \Rightarrow A) \Rightarrow A)$	$\wedge$Int sur (14), (12)
(16)	$(((A \Rightarrow B) \Rightarrow B) \wedge (B \Rightarrow A) \Rightarrow A)$	$\vee$Élim sur (1) de (2) dans (8) et de (9) dans (15)
(17)	$A \vee B \Rightarrow ((A \Rightarrow B) \Rightarrow B) \wedge ((B \Rightarrow A) \Rightarrow A)$	$\Rightarrow$Int de (1) dans (16)

La voici écrite informellement en langage naturel :

Supposons $A \vee B$. Considérons d'abord le cas A. Si on a $A \Rightarrow B$ alors B, ce qui montre $(A \Rightarrow B) \Rightarrow B$; par ailleurs, de A on tire aussi $(B \Rightarrow A) \Rightarrow A$; on a donc montré $((A \Rightarrow B) \Rightarrow B) \wedge ((B \Rightarrow A) \Rightarrow A)$. Le cas B est exactement analogue par symétrie (à l'ordre des conclusions près dans la conjonction finale). Dans les deux cas de la disjonction on a montré $(A \Rightarrow B) \Rightarrow B$. Donc finalement $A \vee B \Rightarrow ((A \Rightarrow B) \Rightarrow B) \wedge ((B \Rightarrow A) \Rightarrow A)$.

En Coq, cette preuve s'écrit :

```
Parameter A B C : Prop.
Theorem thm : A\/B -> ((A->B)->B) /\ ((B->A)->A).
Proof. intro H. destruct H. split. intro H1. apply H1. exact H. intro H2. exact
H. split. intro H1. exact H. intro H2. apply H2. exact H. Qed.
```

On peut aussi directement donner un λ-terme correspondant à la preuve en question :

$$\lambda(h : A \vee B).(\texttt{match } h \texttt{ with } \iota_1 h_0 \mapsto \langle \lambda(h_1 : A \Rightarrow B).\, h_1 h_0,\ \lambda(h_2 : B \Rightarrow A).\, h_0 \rangle,\ \iota_2 h_0 \mapsto \langle \lambda(h_1 : A \Rightarrow B).\, h_0,\ \lambda(h_2 : B \Rightarrow A).\, h_2 h_0 \rangle)$$

(ou en syntaxe Coq : `(fun H : A \/ B => match H with or_introl H0 => conj (fun H1 : A -> B => H1 H0) (fun H2 : B -> A => H0) | or_intror H0 => conj (fun H1 : A -> B => H0) (fun H2 : B -> A => H2 H0) end)`) ✓

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.)

Corrigé. **(1)** Voici une démonstration écrite en informellement en langage naturel : « Supposons $A \vee \neg A$, et on veut montrer $\neg\neg A \Rightarrow A$). Considérons d'abord le cas A : alors certainement $\neg\neg A \Rightarrow A$. Considérons maintenant le cas $\neg A$: alors $\neg\neg A$ aboutit à une contradiction, d'où on peut tirer n'importe quelle conclusion et notamment A, bref, $\neg\neg A \Rightarrow A$ dans ce cas aussi. Dans les deux cas de la disjonction on a montré $\neg\neg A \Rightarrow A$. Donc finalement $(A \vee \neg A) \Rightarrow (\neg\neg A \Rightarrow A)$. »

En Coq, cette preuve s'écrit :

```
Parameter A : Prop.
Theorem thm : (A\/~A) -> (~~A->A).
Proof. intro H. destruct H. split. intro H2. exact H. intro H2. exfalso. apply
H2. exact H. Qed.
```

Revoici la même démonstration écrite comme un λ-terme :

$$\lambda(h : A \vee \neg A).\,(\texttt{match}\ h\ \texttt{with}\ \iota_1 h_0 \mapsto \lambda(h_2 : \neg\neg A).\,h_0,\quad \iota_2 h_1 \mapsto \lambda(h_2 : \neg\neg A).\,\texttt{exfalso}^{(A)}(h_2 h_1))$$

(ou en syntaxe Coq : `(fun H : A \/ ~ A => match H with or_introl H0 => fun H2 : ~ ~ A => H0 | or_intror H1 => fun H2 : ~ ~ A => False_ind A (H2 H1) end))`

(2) Pour prouver l'indémontrabitilité en calcul propositionnel intuitionniste de $(\neg\neg A \Rightarrow A) \Rightarrow (A \vee \neg A)$, on peut utiliser soit une approche sémantique, soit une approche syntaxique. On va expliciter ces différentes approches.

Commençons par l'approche sémantique. Montrons ce résultat avec chacune des sémantiques vues en cours (n'importe laquelle suffit à établir le résultat !).

Une preuve a été donnée en cours basée sur la sémantique des ouverts : si U désigne l'ouvert $]0,1[$ dans $X = \mathbb{R}$, alors $\dot\neg U =]-\infty,0[\cup]1,+\infty[$ donc $\dot\neg\dot\neg U =]0,1[= U$ donc $(\dot\neg\dot\neg U \dot\Rightarrow U) = \mathbb{R}$ tandis que $(U \dot\vee \dot\neg U) = \mathbb{R} \setminus \{0,1\}$, donc finalement $((\dot\neg\dot\neg U \dot\Rightarrow U) \dot\Rightarrow (U \dot\vee \dot\neg U)) = \mathbb{R} \setminus \{0,1\}$; or si $(\neg\neg A \Rightarrow A) \Rightarrow (A \vee \neg A)$ était démontrable en calcul propositionnel intuitionniste, on trouverait X tout entier quel que soit l'ouvert U utilisé pour A (par *correction* de la sémantique des ouverts). Comme ce n'est pas le cas, c'est que la proposition en question n'est pas démontrable.

Une autre preuve est fournie par le cadre de Kripke à trois mondes $\{u,v,w\}$ avec $u \leq v$ et $u \leq w$ (imaginer v,w comme deux futurs possibles de u) et p l'affectation de vérité $(u \mapsto 0, v \mapsto 0, w \mapsto 1)$ (imaginer une vérité encore indécidée et qui pourrait devenir fausse ou vraie), si bien que $\dot\neg p$ est l'affectation de vérité $(u \mapsto 0, v \mapsto 1, w \mapsto 0)$, donc $\dot\neg\dot\neg p$ est l'affectation de vérité $(u \mapsto 0, v \mapsto 0, w \mapsto 1)$ qui est la même que p et $(\dot\neg\dot\neg p \dot\Rightarrow p)$ est l'affectation de vérité constante 1 (i.e., $\dot\top$) ; en revanche, $(p \dot\vee \dot\neg p)$ est l'affectation de vérité $(u \mapsto 0, v \mapsto 1, w \mapsto 1)$, et $((\dot\neg\dot\neg p \dot\Rightarrow p) \dot\Rightarrow (p \dot\vee \dot\neg p))$ est également l'affectation de vérité $(u \mapsto 0, v \mapsto 1, w \mapsto 1)$. Or si $(\neg\neg A \Rightarrow A) \Rightarrow (A \vee \neg A)$ était démontrable en calcul propositionnel intuitionniste, on trouverait constamment 1 quelle que soit l'affectation p utilisé pour A (par *correction* de la sémantique de Kripke). Comme ce n'est pas le cas, c'est que la proposition en question n'est pas démontrable.

Une autre preuve est fournie par la sémantique de la réalisabilité propositionnelle : dans cette sémantique si P est une partie quelconque de $\mathbb{N}$, alors $\dot\neg P$ est $\mathbb{N}$ si P est vide et $\varnothing$ sinon, et $\dot\neg\dot\neg P$ est $\varnothing$ si P est vide et $\mathbb{N}$ sinon. Ainsi, $\dot\neg\dot\neg P \dot\Rightarrow P$ est l'ensemble des programmes qui prennent un entier naturel quelconque en entrée et doivent renvoyer *un élément de P s'il y en a un*. Par contraste, $P \dot\vee \dot\neg P$ est l'ensemble des couples $\langle 0,n\rangle$ avec $n \in P$ ou bien de la forme $\langle 1,n\rangle$ avec n quelconque si $P = \varnothing$. Un élément hypothétique de $((\dot\neg\dot\neg P \dot\Rightarrow P) \dot\Rightarrow (P \dot\vee \dot\neg P))$ pour tous les P à la fois serait un programme qui prend en entrée un élément de $\dot\neg\dot\neg P \dot\Rightarrow P$ et renvoie un élément de $P \dot\vee \dot\neg P$. Or pour $P = \varnothing$ (pour lequel $\dot\neg\dot\neg P \dot\Rightarrow P$ vaut $\mathbb{N}$), ce programme doit renvoyer un couple de la forme $\langle 1,n\rangle$ quelle que soit son entrée ; mais pour $P = \varnothing$, ce même programme doit renvoyer un couple de la forme $\langle 0,n\rangle$ quand on lui passe un élément de $\dot\neg\dot\neg P \dot\Rightarrow P$ (qui est l'ensemble des programmes qui terminent, et en tout cas n'est pas vide) : ceci est contradictoire. Or si $(\neg\neg A \Rightarrow A) \Rightarrow (A \vee \neg A)$ était démontrable en calcul propositionnel intuitionniste, il devrait exister un programme appartenant à $((\dot\neg\dot\neg P \dot\Rightarrow P) \dot\Rightarrow (P \dot\vee \dot\neg P))$ pour tous les P (par *correction* de la sémantique de la réalisabilité propositionnelle). Comme ce n'est pas le cas, c'est que la proposition en question n'est pas démontrable.

Une quatrième preuve est fournie par la sémantique des problèmes finis de Medvedev : considérons les deux problèmes $(\{\bullet\},\{\bullet\})$ et $(\{\bullet\},\varnothing)$ (qui ont le même ensemble de candidats, et qui sont échangés par $\dot\neg$). Sur le premier, $\dot\neg\dot\neg P \dot\Rightarrow P$ vaut $(\{\bullet\},\{\bullet\})$ (où on note abusivement $\bullet$ pour l'unique fonction $\{\bullet\} \to \{\bullet\}$) tandis que $P \dot\vee \dot\neg P$ vaut $(\{\bullet_1,\bullet_2\},\{\bullet_1\})$, donc $((\dot\neg\dot\neg P \dot\Rightarrow P) \dot\Rightarrow (P \dot\vee \dot\neg P))$ vaut $(\{\bullet_1,\bullet_2\},\{\bullet_1\})$ (où on note abusivement $\bullet_i$ pour l'unique fonction $\{\bullet\} \to \{\bullet_1,\bullet_2\}$ envoyant $\bullet$ sur $\bullet_i$). Sur le second problème, $\dot\neg\dot\neg P \dot\Rightarrow P$ vaut $(\{\bullet\},\{\bullet\})$ tandis que $P \dot\vee \dot\neg P$ vaut

$(\{\bullet_1, \bullet_2\}, \{\bullet_2\})$, donc $((\dot{\neg}\dot{\neg} P \dot{\Rightarrow} P) \dot{\Rightarrow} (P \dot{\vee} \dot{\neg} P))$ vaut $(\{\bullet_1, \bullet_2\}, \{\bullet_2\})$. Ces deux ensembles de solutions sont disjoints, donc il n'y a pas de solution commune. Or si $(\neg\neg A \Rightarrow A) \Rightarrow (A \vee \neg A)$ était démontrable en calcul propositionnel intuitionniste, il devrait exister une solution appartenant à $((\dot{\neg}\dot{\neg} P \dot{\Rightarrow} P) \dot{\Rightarrow} (P \dot{\vee} \dot{\neg} P))$ pour tous les P ayant un même ensemble de candidats (par *correction* de la sémantique de la sémantique de Medvedev). Comme ce n'est pas le cas, c'est que la proposition en question n'est pas démontrable.

Outre ces quatre preuves sémantiques, on peut aussi prouver l'idémontrabilité de $(\neg\neg A \Rightarrow A) \Rightarrow (A \vee \neg A)$ en calcul propositionnel intuitionniste de façon syntaxique, par la recherche d'une démonstration sans coupure, comme suit.

Puisque $\vdash (\neg\neg A \Rightarrow A) \Rightarrow (A \vee \neg A)$ si et seulement si $\neg\neg A \Rightarrow A \vdash A \vee \neg A$ (par les règles d'introduction et d'élimination du $\Rightarrow$ en déduction naturelle), il suffit de montrer qu'on n'a pas $\neg\neg A \Rightarrow A \vdash A \vee \neg A$. Par l'existence d'une démonstration sans coupure (ou plus précisément, la propriété de la sous-formule), si on avait ce séquent, il y en aurait une démonstration dans laquelle toute formule qui apparaît est l'une des six suivantes : $\bot, A, \neg A, \neg\neg A, A \vee \neg A, \neg\neg A \Rightarrow A$. Il s'agit donc de considérer tous les séquents ayant un sous-ensemble de ces six formules comme hypothèses et une de ces six formules comme conclusion : cela fait $2^6 \times 6 = 384$ possibilités, un peu fastidieux à lister complètement à la main, mais on peut se simplifier la tâcher en considérant les ensembles suivants (tous facilement démontrables) :

— ceux ayant la conclusion parmi les hypothèses,
— ceux ayant $\bot$ dans les hypothèses, ou bien à la fois A et $\neg A$, ou bien à la fois $\neg A$ et $\neg\neg A$, et une conclusion quelconque,
— $\Gamma, A \vdash \neg\neg A$ (où Γ est quelconque),
— $\Gamma, \neg\neg A, (A \vee \neg A) \vdash A$,
— $\Gamma, \neg\neg A, (\neg\neg A \Rightarrow A) \vdash A$,
— $\Gamma, A \vdash (A \vee \neg A)$,
— $\Gamma, \neg A \vdash (A \vee \neg A)$.
— $\Gamma, \neg\neg A, (\neg\neg A \Rightarrow A) \vdash A \vee \neg A$,
— $\Gamma, A \vdash (\neg\neg A \Rightarrow A)$.
— $\Gamma, \neg A \vdash (\neg\neg A \Rightarrow A)$.

On peut ensuite se convaincre en examinant chaque règle de la logique (en calcul des séquents) qu'aucune application d'aucune règle à un de ces séquents ne donne de séquent nouveau (parmi ceux dont les hypothèses et la conclusion sont dans les six formules listées !). Les séquents qu'on vient de lister sont donc exactement tous les séquents valables dont les hypothèses et la conclusion sont dans les six formules listées. Comme $\neg\neg A \Rightarrow A \vdash A \vee \neg A$ n'en fait pas partie, il n'est pas valable, pas plus que $\vdash (\neg\neg A \Rightarrow A) \Rightarrow (A \vee \neg A)$.

(Cett approche syntaxique est nettement plus pénible que les approches sémantiques qu'on a vues. Elle a cependant l'avantage de relever d'un algorithme systématique.)

(3) Observons d'abord que si on postule $P \vee \neg P$ pour toute proposition P, alors on peut en déduire $\neg\neg Q \Rightarrow Q$ pour toute proposition Q : ce sens-là est évident, car $\neg\neg Q \Rightarrow Q$ découle de $Q \vee \neg Q$ comme expliqué dans la question (1) (et en utilisant implicitement le fait qu'on peut substituer une proposition quelconque à une variable propositionnelle).

Reste à traiter l'autre sens, i.e., montrer que si on postule $\neg\neg Q \Rightarrow Q$ pour toute proposition Q, alors on peut en déduire $P \vee \neg P$ pour toute proposition P. Soit donc P une proposition quelconque (ou une variable propositionnelle, si on préfère). Or $\neg(P \vee \neg P)$ équivaut à $\neg P \wedge \neg\neg P$ (ceci est une application de la tautologie $((A \vee B) \Rightarrow C) \Leftrightarrow (A \Rightarrow C) \wedge (B \Rightarrow C)$ en remplaçant A par P, B par $\neg P$ et C par $\bot$) ; mais clairement $\neg P \wedge \neg\neg P$ implique $\bot$ (ceci est une application de la tautologie $A \wedge (A \Rightarrow B) \Rightarrow B$ en remplaçant A par $\neg P$ et B par $\bot$) : donc on a montré $\neg\neg(P \vee \neg P)$ (sans hypothèse). Si on postule $\neg\neg Q \Rightarrow Q$, il n'y a qu'à appliquer ce fait avec Q valant $P \vee \neg P$ pour conclure $P \vee \neg P$. ✓

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.

Corrigé. Voici une preuve écrite informellement en langage naturel :

Supposons $\neg\neg A \wedge \neg\neg B$ (donc à la fois $\neg\neg A$ et $\neg\neg B$), et on veut montrer $\neg\neg(A \wedge B)$. Pour ça, supposons $\neg(A \wedge B)$ et on veut arriver à une contradiction. Supposons provisoirement qu'on ait A. Si de plus on a B, alors on a $A \wedge B$, ce qui contredit $\neg(A \wedge B)$: ceci montre $\neg B$ (toujours sous l'hypothèse A !) ; mais commme on a $\neg\neg B$, on a une contradiction. On a donc prouvé $\neg A$; mais comme on a $\neg\neg A$, on a une contradiction.

La revoici écrite dans le style « drapeau » :

(1) $\boxed{\neg\neg A \wedge \neg\neg B}$
(2) $\quad\boxed{\neg(A \wedge B)}$

(3)	$\neg\neg A$	$\wedge$Élim$_1$ sur (1)
(4)	$\boxed{A}$	
(5)	$\neg\neg B$	$\wedge$Élim$_2$ sur (1)
(6)	$\boxed{B}$	
(7)	$A \wedge B$	$\wedge$Int sur (4), (6)
(8)	$\bot$	$\Rightarrow$Élim sur (2) et (7)
(9)	$\neg B$	$\Rightarrow$Int de (6) dans (8)
(10)	$\bot$	$\Rightarrow$Élim sur (5) et (9)
(11)	$\neg A$	$\Rightarrow$Int de (4) dans (10)
(12)	$\bot$	$\Rightarrow$Élim sur (3) et (11)
(13)	$\neg\neg(A \wedge B)$	$\Rightarrow$Int de (2) dans (12)
(14)	$\neg\neg A \wedge \neg\neg B \Rightarrow \neg\neg(A \wedge B)$	$\Rightarrow$Int de (1) dans (13)

La voici réécrite en forme d'arbre de preuve (en omettant certaines hypothèses superflues à gauche du symbole '$\vdash$', ou, ce qui revient au même, en faisant un usage tacite de la règle d'affaiblissement) :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg\neg A \wedge \neg\neg B \vdash \neg\neg A \wedge \neg\neg B}\,\text{Ax}}{\neg\neg A \wedge \neg\neg B \vdash \neg\neg A}\,\wedge\text{Élim}_1 \qquad \dfrac{\dfrac{\dfrac{}{\neg\neg A \wedge \neg\neg B \vdash \neg\neg A \wedge \neg\neg B}\,\text{Ax}}{\neg\neg A \wedge \neg\neg B \vdash \neg\neg B}\,\wedge\text{Élim}_2 \qquad \dfrac{\dfrac{\dfrac{}{\neg(A \wedge B) \vdash \neg(A \wedge B)}\,\text{Ax} \qquad \dfrac{\dfrac{}{A \vdash A}\,\text{Ax} \quad \dfrac{}{B \vdash B}\,\text{Ax}}{A, B \vdash A \wedge B}\,\wedge\text{Int}}{\neg(A \wedge B), A, B \vdash \bot}\,\Rightarrow\text{Élim}}{\neg(A \wedge B), A \vdash \neg B}\,\Rightarrow\text{Int}}{\neg\neg A \wedge \neg\neg B, \neg(A \wedge B), A \vdash \bot}\,\Rightarrow\text{Élim}}{\neg\neg A \wedge \neg\neg B, \neg(A \wedge B) \vdash \neg A}\,\Rightarrow\text{Int}}{\neg\neg A \wedge \neg\neg B, \neg(A \wedge B) \vdash \bot}\,\Rightarrow\text{Élim}}{\neg\neg A \wedge \neg\neg B \vdash \neg\neg(A \wedge B)}\,\Rightarrow\text{Int}}{\vdash \neg\neg A \wedge \neg\neg B \Rightarrow \neg\neg(A \wedge B)}\,\Rightarrow\text{Int}$$

Ou, avec uniquement les conclusions de chaque séquent (mais en indiquant, en contrepartie, à chaque décharge d'hypothèse, où cette hypothèse est déchargée : les lettres h, k, h_1, h_2 sont de tels renvois, et on a choisi les mêmes lettres que dans le terme de preuve écrit ci-dessous pour aider à la comparaison) :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{h}{\neg\neg A \wedge \neg\neg B}}{\neg\neg A}\,\wedge\text{Élim}_1 \qquad \dfrac{\dfrac{\dfrac{h}{\neg\neg A \wedge \neg\neg B}}{\neg\neg B}\,\wedge\text{Élim}_2 \qquad \dfrac{\dfrac{\dfrac{k}{\neg(A \wedge B)} \qquad \dfrac{\overset{h_1}{\overline{A}} \quad \overset{h_2}{\overline{B}}}{A \wedge B}\,\wedge\text{Int}}{\bot}\,\Rightarrow\text{Élim}}{\neg B}\,\Rightarrow\text{Int}(h_2)}{\bot}\,\Rightarrow\text{Élim}}{\neg A}\,\Rightarrow\text{Int}(h_1)}{\bot}\,\Rightarrow\text{Élim}}{\neg\neg(A \wedge B)}\,\Rightarrow\text{Int}(k)}{\neg\neg A \wedge \neg\neg B \Rightarrow \neg\neg(A \wedge B)}\,\Rightarrow\text{Int}(h)$$

En Coq, cette preuve s'écrit :

```
Parameter A : Prop. Parameter B : Prop.
Theorem thm : ~~A /\ ~~B -> ~~(A/\B).
Proof. intro H. intro K. destruct H as [H1w H2w]. apply H1w. intro H1. apply
H2w. intro H2. apply K. split. exact H1. exact H2. Qed.
```

En voici un λ-terme de preuve :

$$\lambda(h : \neg\neg A \wedge \neg\neg B).\, \lambda(k : \neg(A \wedge B)).\, (\pi_1 h)(\lambda(h_1 : a).\, (\pi_2 h)(\lambda(h_2 : B).\, k\langle h_1, h_2\rangle))$$

(ou en syntaxe Coq : `fun (H : ~ ~ A /\ ~ ~ B) (K : ~ (A /\ B)) => match H with conj H1w H2w => H1w (fun H1 : A => H2w (fun H2 : B => K (conj H1 H2))) end`; la forme est un peu différente parce que Coq utilise un `match` pour déstructurer une conjonction alors que nous avons utilisé π_1 et π_2, mais une fois noté que $\pi_1 h$ et $\pi_2 h$ correspondent à `H1w` et `H2w` c'est bien essentiellement le même 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.

Corrigé. Montrons d'abord que si on postule $\neg P \vee \neg\neg P$ pour toute proposition P, on peut en déduire $\neg(Q \wedge R) \Rightarrow \neg Q \vee \neg R$ pour toutes propositions Q, R. Soient donc Q, R deux propositions quelconques (ou variables propositionnelles, si on préfère), et on veut prouver $\neg(Q \wedge R) \Rightarrow \neg Q \vee \neg R$. Supposons $\neg(Q \wedge R)$ et on veut prouver $\neg Q \vee \neg R$. Appliquons notre postulat $\neg P \vee \neg\neg P$ avec P valant Q puis R successivement : on a $\neg Q \vee \neg\neg Q$ et $\neg R \vee \neg\neg R$; par la loi d'élimination du $\vee$, ceci nous permet de raisonner par cas (dans chaque cas, on cherche à prouver $\neg Q \vee \neg R$) : dans le cas où $\neg Q$, ainsi que dans le cas où $\neg R$, on a évidemment $\neg Q \vee \neg R$; il reste donc à traiter le cas où $\neg\neg Q$ et $\neg\neg R$. Or $\neg\neg A \wedge \neg\neg B \Rightarrow \neg\neg(A \wedge B)$ est démontrable en calcul propositionnel intuitionniste (exercice 3.6). Donc on peut affirmer $\neg\neg(Q \wedge R)$, ce qui contredit $\neg(Q \wedge R)$, et une contradiction permet de tirer n'importe quelle conclusion, notamment $\neg Q \vee \neg R$. Bref, dans chacun des cas on a bien prouvé $\neg Q \vee \neg R$. On peut maintenant décharger la supposition $\neg(Q \wedge R)$ et affirmer que $\neg(Q \wedge R) \Rightarrow \neg Q \vee \neg R$ comme on voulait.

Réciproquement, montrons que si on postule $\neg(Q \wedge R) \Rightarrow \neg Q \vee \neg R$ pour toutes propositions Q, R, on peut en déduire $\neg P \vee \neg\neg P$ pour toute proposition P. Soit donc P une proposition quelconque (ou variable propositionnelle, si on préfère), et on veut prouver $\neg P \vee \neg\neg P$. Appliquons notre postulat $\neg(Q \wedge R) \Rightarrow \neg Q \vee \neg R$ avec Q valant P et R valant $\neg P$: il nous donne $\neg(P \wedge \neg P) \Rightarrow \neg P \vee \neg\neg P$. Or $\neg(P \wedge \neg P)$ est évident (une preuve en est donnée par le λ-terme $\lambda(h : P \wedge \neg P).\,(\pi_2 h)(\pi_1 h)$), donc on a bien $\neg P \vee \neg\neg P$ comme ou voulait.

Si $\neg(A \wedge B) \Rightarrow \neg A \vee \neg B$ était démontrable en calcul propositionnel intuitionniste, on pourrait remplacer les variables propositionnelles A, B par des propositions Q, R quelconques. Notamment, d'après ce qu'on vient de voir au paragraphe précédent, on pourrait démontrer $\neg P \vee \neg\neg P$ pour P une proposition quelconque. Mais $\neg C \vee \neg\neg C$ n'est certainement pas prouvable en calcul propositionnel intuitionniste : car s'il l'était, par la propriété de la disjonction, $\neg C$ ou bien $\neg\neg C$ le serait, ce qui n'est pas le cas (même classiquement, on ne peut pas prouver $\neg C$ seul, ni $\neg\neg C$ seul, qui peut prendre l'une ou l'autre valeur de vérité). ✓

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).

Corrigé. On prend bien garde au fait qu'il faut parenthéser comme $\lambda k.\, k\,((k\,(\lambda x.x))\,(\lambda y.y))$.

Dans une première phase, on collecte les types et contraintes suivants : $k : \eta_1$, $x : \eta_2$ donc $\lambda x.x : \eta_2 \to \eta_2$, puis $k(\lambda x.x) : \eta_3$ avec $\eta_1 = ((\eta_2 \to \eta_2) \to \eta_3)$, puis $y : \eta_4$ donc $\lambda y.y : \eta_4 \to \eta_4$, puis $k(\lambda x.x)(\lambda y.y) : \eta_5$ avec $\eta_3 = ((\eta_4 \to \eta_4) \to \eta_5)$, puis $k(k(\lambda x.x)(\lambda y.y)) : \eta_6$ avec $\eta_1 = (\eta_5 \to \eta_6)$; et le type final est $\eta_1 \to \eta_6$. On a donc les contraintes suivantes :

$$\eta_1 = ((\eta_2 \to \eta_2) \to \eta_3)$$
$$\eta_3 = ((\eta_4 \to \eta_4) \to \eta_5)$$
$$\eta_1 = (\eta_5 \to \eta_6)$$

On examine d'abord la contrainte $\eta_1 = ((\eta_2 \to \eta_2) \to \eta_3)$: ici, 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 $\eta_1 \mapsto ((\eta_2 \to \eta_2) \to \eta_3)$ dans la substitution et

l'appliquer aux contraintes qui deviennent donc :

$$\eta_3 = ((\eta_4 \to \eta_4) \to \eta_5)$$
$$((\eta_2 \to \eta_2) \to \eta_3) = (\eta_5 \to \eta_6)$$

On examine ensuite la contrainte $\eta_3 = ((\eta_4 \to \eta_4) \to \eta_5)$: de nouveau, η_3 est une variable de type, et elle n'apparaît pas dans l'autre membre de la contrainte : on enregistre $\eta_3 \mapsto ((\eta_4 \to \eta_4) \to \eta_5)$ et on l'applique à la substitution déjà enregistrée qui devient donc $\eta_1 \mapsto ((\eta_2 \to \eta_2) \to (\eta_4 \to \eta_4) \to \eta_5)$, ainsi qu'à la contrainte restante. Cette dernière contrainte à examiner est :

$$((\eta_2 \to \eta_2) \to (\eta_4 \to \eta_4) \to \eta_5) = (\eta_5 \to \eta_6)$$

Cette fois-ci les deux membres sont des types complexes, donc elle se transforme en deux contraintes (attention au parenthésage !) :

$$(\eta_2 \to \eta_2) = \eta_5$$
$$((\eta_4 \to \eta_4) \to \eta_5) = \eta_6$$

On examine à présent $(\eta_2 \to \eta_2) = \eta_5$: ici, η_5 est une variable de type, et elle n'apparaît pas dans l'autre membre de la contrainte : on enregistre $\eta_5 \mapsto (\eta_2 \to \eta_2)$, et on applique cette substitution à ce qui reste de contrainte (qui devient donc $((\eta_4 \to \eta_4) \to \eta_2 \to \eta_2) = \eta_6$) et à ce qu'on a déjà déterminé comme substitution (qui devient donc $\eta_1 \mapsto ((\eta_2 \to \eta_2) \to (\eta_4 \to \eta_4) \to \eta_2 \to \eta_2)$ et $\eta_3 \mapsto ((\eta_4 \to \eta_4) \to \eta_2 \to \eta_2)$). Il reste enfin la contrainte $((\eta_4 \to \eta_4) \to \eta_2 \to \eta_2) = \eta_6$: de nouveau, η_6 est une variable de type, et elle n'apparaît pas dans l'autre membre de la contrainte : on enregistre donc $\eta_6 \mapsto ((\eta_4 \to \eta_4) \to \eta_2 \to \eta_2)$, et on a trouvé la substitution finale :

$$\eta_1 \mapsto ((\eta_2 \to \eta_2) \to (\eta_4 \to \eta_4) \to \eta_2 \to \eta_2)$$
$$\eta_3 \mapsto ((\eta_4 \to \eta_4) \to \eta_2 \to \eta_2)$$
$$\eta_5 \mapsto (\eta_2 \to \eta_2)$$
$$\eta_6 \mapsto ((\eta_4 \to \eta_4) \to \eta_2 \to \eta_2)$$

Il ne reste plus qu'à l'appliquer à tous les types intermédiaires et au type final, ce qui donne

$$\begin{array}{rl} & \lambda(k : (\eta_2 \to \eta_2) \to (\eta_4 \to \eta_4) \to \eta_2 \to \eta_2).\, k\,(k\,(\lambda(x : \eta_2).x)\,(\lambda(y : \eta_4).y)) \\ : & ((\eta_2 \to \eta_2) \to (\eta_4 \to \eta_4) \to \eta_2 \to \eta_2) \to (\eta_4 \to \eta_4) \to \eta_2 \to \eta_2 \end{array}$$

ou, si on veut renommer les variables restantes pour être un peu plus lisible

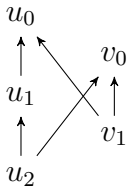
$$\begin{array}{rl} & \lambda(k : (\alpha \to \alpha) \to (\beta \to \beta) \to \alpha \to \alpha).\, k\,(k\,(\lambda(x : \alpha).x)\,(\lambda(y : \beta).y)) \\ : & ((\alpha \to \alpha) \to (\beta \to \beta) \to \alpha \to \alpha) \to (\beta \to \beta) \to \alpha \to \alpha \end{array}$$

comme λ-terme annoté et comme type final. ✓

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 $((\dot\neg\,\dot\neg\, p \mathrel{\dot\Rightarrow} p) \mathrel{\dot\Rightarrow} (p \mathbin{\dot\vee} \dot\neg\, p)) \mathrel{\dot\Rightarrow} (\dot\neg\, p \mathbin{\dot\vee} \dot\neg\,\dot\neg\, p)$, 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.

Corrigé. On calcule successivement l'affectation de vérité de chacune des sous-formules de la formule proposée : on se rappelle que $q \Rightarrow r$ vaut 1 en un monde w lorsque dans tout monde w' accessible depuis w pour lequel $q(w') = 1$ on a aussi $r(w') = 1$, et notamment, $\neg q$ vaut q en un monde w lorsque dans tout monde w' accessible depuis w on a $q(w') = 0$. Par ailleurs, pour éviter de se tromper, on vérifie à tout stade que les affectations de vérité sont permanentes, c'est-à-dire que si $q(w) = 1$ on a $q(w') = 1$ pour tout monde w' accessible depuis w. On obtient les résultats tabulés ci-dessous :

Formule	u_2	v_1	u_1	v_0	u_0
p	0	0	0	0	1
$\neg p$	0	0	0	1	0
$p \vee \neg p$	0	0	0	1	1
$\neg\neg p$	0	0	1	0	1
$\neg p \vee \neg\neg p$	0	0	1	1	1
$\neg\neg p \Rightarrow p$	0	1	0	1	1
$(\neg\neg p \Rightarrow p) \Rightarrow (p \vee \neg p)$	1	0	1	1	1
$((\neg\neg p \Rightarrow p) \Rightarrow (p \vee \neg p)) \Rightarrow (\neg p \vee \neg\neg p)$	0	1	1	1	1

On constate que l'affecatation de vérité de $((\neg\neg p \Rightarrow p) \Rightarrow (p \vee \neg p)) \Rightarrow (\neg p \vee \neg\neg p)$ n'est pas identiquement 1 sur le cadre. Or si $((\neg\neg A \Rightarrow A) \Rightarrow (A \vee \neg A)) \Rightarrow (\neg A \vee \neg\neg A)$ était démontrable en calcul propositionnel intuitionniste, on trouverait constamment 1 dans tout cadre et en remplaçant p par n'importe quelle affectation de vérité sur le cadre (par *correction* de la sémantique de la sémantique de Kripke) : c'est donc que cette formule n'est pas démontrable. ✓

Exercice 4.2. ($\star\star$)

Pour $n \in \mathbb{N}$, on considère le cadre de Kripke $\{w_0, \dots, w_{n-1}\}$ 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 $p_0, \dots, p_n$ 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 $p_0, \dots, p_n$.

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.

Corrigé. La contrainte sur une affectation de vérité sur un cadre de Kripke W est d'être permanente, i.e., si $p(w) = 1$ alors $p(w') = 1$ pour tout monde w' accessible depuis w (c'est-à-dire $w \leq w'$), autrement dit d'etre une fonction croissante $W \to \{0,1\}$. Comme ici W est simplement $\{0, \dots, n-1\}$ avec l'ordre inversé, les seules fonctions décroissantes $\{0, \dots, n-1\} \to \{0,1\}$ étant celles qui valent 1 jusqu'à un certain point et 0 ensuite, on obtient les p_k qu'on a décrits.

Le calcul de $\dot\wedge$ et $\dot\vee$ est facile puisqu'il se fait monde par monde : on a $(p_k \mathbin{\dot\wedge} p_\ell)(w_i) = 1$ lorsque $p_k(w_i) = 1$ et $p_\ell(w_i) = 1$, c'est-à-dire $i < k$ et $i < \ell$, ce qui équivaut à $i < \min(k, \ell)$, ce qui montre $p_k \mathbin{\dot\wedge} p_\ell = p_{\min(k,\ell)}$; de même, on a $(p_k \mathbin{\dot\vee} p_\ell)(w_i) = 1$ lorsque $p_k(w_i) = 1$ ou $p_\ell(w_i) = 1$, c'est-à-dire $i < k$ ou $i < \ell$, ce qui équivaut à $i < \max(k, \ell)$, ce qui montre $p_k \mathbin{\dot\vee} p_\ell = p_{\max(k,\ell)}$.

Reste à traiter le cas de $\dot\Rightarrow$. Si $k \leq \ell$, alors $p_k(w_j) = 1$ implique $p_\ell(w_j) = 1$ (car $j < k$ implique $j < \ell$), et ce, pour tout j, ce qui montre $(p_k \mathrel{\dot\Rightarrow} p_\ell) = \dot\top = p_n$ dans ce cas. En revanche, si $k > \ell$, il faut distinguer deux cas : lorsque $j < \ell$, on a $p_k(w_j) = 1$ et $p_\ell(w_j) = 1$, ce qui montre que $(p_k \mathrel{\dot\Rightarrow} p_\ell)(w_i) = 1$ pour $i < \ell$; mais comme $p_k(w_\ell) = 1$ et cependant $p_\ell(w_\ell) = 0$, on a $(p_k \mathrel{\dot\Rightarrow} p_\ell)(w_i) = 0$ pour $i \geq \ell$; c'est-à-dire que $(p_k \mathrel{\dot\Rightarrow} p_\ell) = p_\ell$ dans ce cas. Enfin, $\dot\neg\, q$ est simplement $q \mathrel{\dot\Rightarrow} \dot\bot$, c'est-à-dire $q \mathrel{\dot\Rightarrow} p_0$.

Pour résumer :

$$\begin{aligned} p_k \mathbin{\dot\wedge} p_\ell &= p_{\min(k,\ell)} \\ p_k \mathbin{\dot\vee} p_\ell &= p_{\max(k,\ell)} \\ (p_k \dot\Rightarrow p_\ell) &= \begin{cases} p_n & \text{si } k \leq \ell \\ p_\ell & \text{si } k > \ell \end{cases} \\ \dot\neg p_k &= \begin{cases} p_n & \text{si } k = 0 \\ p_0 & \text{si } k > 0 \end{cases} \end{aligned}$$

La formule $A \vee \neg A$, classiquement démontrable, n'est pas validée par ce cadre lorsque $n \geq 2$ (prendre p_1 pour A : on trouve $p_1 \mathbin{\dot\vee} \dot\neg p_1 = p_1$). La formule $\neg A \vee \neg\neg A$, en revanche, est validée par ce cadre (car $\dot\neg \dot\neg p_k = p_n$ si $k > 0$ et p_0 si $k = 0$) mais n'est pas prouvable intuitionnistement (par la propriété de la disjonction : si elle était prouvable, soit $\neg A$ soit $\neg\neg A$ le serait, or elles ne sont même pas prouvables classiquement). ✓

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 $(\dot\neg A_z \dot\Rightarrow B_z \mathbin{\dot\vee} C_z)$ auquel appliquer r.)

Corrigé. Supposons par l'absurde qu'il existe (un *même*) r qui réalise $(\neg A \Rightarrow B \vee C) \Rightarrow (\neg A \Rightarrow B) \vee (\neg A \Rightarrow C)$, c'est-à-dire qui appartienne à $(\dot\neg A \dot\Rightarrow B \mathbin{\dot\vee} C) \dot\Rightarrow (\dot\neg A \dot\Rightarrow B) \mathbin{\dot\vee} (\dot\neg A \dot\Rightarrow C)$ quels que soient $A, B, C \subseteq \mathbb{N}$ (où $P \mathbin{\dot\vee} Q = \{\langle 0, m\rangle : m \in P\} \cup \{\langle 1, n\rangle : n \in Q\}$ et $(P \dot\Rightarrow Q) = \{e \in \mathbb{N} : \varphi_e(P)\!\downarrow \subseteq Q\}$, et on se rappelle bien sûr que $\dot\neg P = (P \dot\Rightarrow \varnothing)$ vaut $\mathbb{N}$ si P est vide et $\varnothing$ si P n'est pas vide).

On pose $L := \{\langle e, x\rangle : \varphi_e(x)\!\downarrow = 1\}$ et $M := \{\langle e, x\rangle : \varphi_e(x)\!\downarrow = 2\}$. On a vu dans l'exercice 1.15 qu'il n'existe aucun programme s qui, prenant en entrée un $z \in \mathbb{N}$, termine toujours en temps fini, et renvoie « vrai » lorsque $z \in L$ et « faux » lorsque $z \in M$ (et une réponse quelconque, lorsque $z \notin L \cup M$, mais le programme doit quand même terminer). Or on va utiliser r pour faire précisément une telle chose, ce qui constituera une contradiction.

Définissons les ensembles A_z, B_z, C_z pour $z = \langle e, x\rangle$ comme suit :

$$\begin{aligned} A_z &= \varnothing \text{ si } \varphi_e(x)\!\downarrow = 1 \text{ ou } \varphi_e(x)\!\downarrow = 2, \\ &= \mathbb{N} \text{ sinon} \\ (\text{donc } \dot\neg A_z &= \mathbb{N} \text{ si } \varphi_e(x)\!\downarrow = 1 \text{ ou } \varphi_e(x)\!\downarrow = 2, \\ &= \varnothing \text{ sinon}) \\ B_z &= \mathbb{N} \text{ si } \varphi_e(x)\!\downarrow = 1, \\ &= \varnothing \text{ sinon} \\ C_z &= \mathbb{N} \text{ si } \varphi_e(x)\!\downarrow = 2, \\ &= \varnothing \text{ sinon} \end{aligned}$$

Donné $z \in \mathbb{N}$, considérons le programme p_z défini comme suit. Il prend un unique argument, qu'il ignore. Il décode le couple $z = \langle e, x\rangle$, puis il exécute $\varphi_e(x)$ (au moyen d'un interpréteur universel). Si $\varphi_e(x)\!\downarrow = 1$, il renvoie $\langle 0, 0\rangle$, et si $\varphi_e(x)\!\downarrow = 2$, il renvoie $\langle 1, 0\rangle$; dans tout autre cas, il fait une boucle infinie (y compris bien sûr si $\varphi_e(x)\!\uparrow$, c'est-à-dire si l'exécution ne termine jamais : dans ce cas forcément p_z ne termine pas non plus).

On remarque que p_z est construit algorithmiquement en fonction de z (par le théorème s-m-n si on veut).

Par construction, si on passe à p_z un argument de $\dot\neg A_z$, ce qui implique que $\varphi_e(x)\!\downarrow = 1$ ou $\varphi_e(x)\!\downarrow = 2$ (l'argument est une *promesse* de ce fait), alors il renvoie un élément de $B_z \mathbin{\dot\vee} C_z$ (à savoir un couple $\langle 0, n\rangle$ avec $n \in B_z$ ou $\langle 1, n\rangle$ avec $n \in C_z$). On a donc $p_z \in (\dot\neg A_z \dot\Rightarrow B_z \mathbin{\dot\vee} C_z)$.

Par hypothèse sur r, on doit donc avoir $\varphi_r(p_z)$ défini et appartenant à $(\dot\neg A_z \dot\Rightarrow B_z) \mathbin{\dot\vee} (\dot\neg A_z \dot\Rightarrow C_z)$. Notamment, $\varphi_r(p_z)$ est (le codage d')un couple dont la première coordonnée vaut 0 ou 1 et indique si la seconde est dans $\dot\neg A_z \dot\Rightarrow B_z$

ou dans $\dot{\neg} A_z \dot{\Rightarrow} C_z$. On considère le programme s qui prend un z en entrée, calcule $\varphi_r(p_z)$ (noter que ce calcul est bien algorithmique puisque p_z est construit algorithmiquement en fonction de z, et il termine toujours d'après ce qu'on vient de dire), et renvoie « vrai » si la première coordonnée du résultat vaut 0 et « faux » si la seconde coordonnée du résultat vaut 1.

Si $z = \langle e, x\rangle$ avec $\varphi_e(x)\!\downarrow = 1$, alors $C_z = \varnothing$ et $\dot{\neg} A_z = \mathbb{N}$, donc $(\dot{\neg} A_z \dot{\Rightarrow} C_z) = \varnothing$, donc $\varphi_r(p_z)$ doit forcément être de la forme $\langle 0, \cdots\rangle$, et s renvoie « vrai » sur l'entrée z. Symétriquement, si $z = \langle e, x\rangle$ avec $\varphi_e(x)\!\downarrow = 2$, alors $(\dot{\neg} A_z \dot{\Rightarrow} B_z) = \varnothing$ donc $\varphi_r(p_z)$ doit forcément être de la forme $\langle 1, \cdots\rangle$, et s renvoie « faux » sur l'entrée z.

Donc s termine toujours, et sépare L et M puisqu'il renvoie « vrai » sur le premier et « faux » sur le second. Ceci contredit le fait que L et M sont récursivement inséparables. On a abouti à une contradiction : c'est qu'en fait r n'existait pas.

(La formule de Kreisel-Putnam n'est donc pas réalisable. En particulier, par *correction* de la sémantique de la réalisabilité, elle n'est pas démontrable dans le calcul propositionnel intuitionniste, c'est-à-dire, par Curry-Howard, qu'il n'y a pas de terme du λ-calcul simplement typé ayant pour type $(\neg A \Rightarrow B \vee C) \Rightarrow (\neg A \Rightarrow B) \vee (\neg A \Rightarrow C)$.) ✓

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.)

Corrigé. **(1)** Soit (X, S) un problème fini. On rappelle que $\dot{\bot} = (\{\bullet\}, \varnothing)$. Le problème $\dot{\neg}(X, S)$, c'est-à-dire $(X, S) \dot{\Rightarrow} (\{\bullet\}, \varnothing)$ a une seul candidat, à savoir l'unique fonction $X \to \{\bullet\}$, qu'on notera abusivement aussi $\bullet$; les solutions de $\dot{\neg}(X, S)$ sont les fonctions $X \to \{\bullet\}$ qui envoient chaque élément de S dans $\varnothing$: or si $S = \varnothing$ c'est le cas de l'unique fonction $\bullet$, tandis que si $S \neq \varnothing$, elle n'envoie pas S dans $\varnothing$. Pour résumer, $\dot{\neg}(X, S)$ a toujours exactement un candidat $\bullet$, et ce candidat est solution exactement lorsque le problème (X, S) de départ a n'a pas de solution.

(2) Considérons d'abord $\dot{\neg} A \dot{\Rightarrow} B \dot{\vee} C$ en remplaçant A, B, C par trois problèmes finis (X, S), (Y, T) et (Z, U) respectivement. On a vu que $\dot{\neg} A$ a toujours exactement un candidat $\bullet$, et qu'il est solution précisément lorsque $S = \varnothing$, et sinon aucune. Les candidats de $\dot{\neg} A \dot{\Rightarrow} B \dot{\vee} C$ sont donc des fonctions de $\{\bullet\}$ vers l'ensemble des candidats $Y \uplus Z$ de $B \dot{\vee} C$, qu'on peut donc identifier à $Y \uplus Z$ (en identifiant une fonction $\{\bullet\} \to V$, pour V un ensemble quelconque, avec l'image de $\bullet$ par cette fonction). Parmi ces candidats, si $S \neq \varnothing$, ils sont tous solutions (car $\dot{\neg} A$ n'a pas de solution donc il n'y a pas de contrainte), tandis que si $S = \varnothing$, ceux qui sont solution sont ceux solution de B et de C, i.e., c'est $T \uplus U$.

Or la même description vaut pour $(\dot{\neg} A \dot{\Rightarrow} B) \dot{\vee} (\dot{\neg} A \dot{\Rightarrow} C)$: le problème $(\dot{\neg} A \dot{\Rightarrow} B)$ a un ensemble de candidats qui s'identifie à celui Y de B, et si $S \neq \varnothing$ ils sont tous solutions tandis que si $S = \varnothing$ ceux qui sont solutions sont les solutions de B, c'est-à-dire T ; et le problème $(\dot{\neg} A \dot{\Rightarrow} C)$ a un ensemble de candidats qui s'identifie à celui Z de C, et si $S \neq \varnothing$ ils sont tous solutions tandis que si $S = \varnothing$ ceux qui sont solutions sont les solutions de C, c'est-à-dire V. Bref, $(\dot{\neg} A \dot{\Rightarrow} B) \dot{\vee} (\dot{\neg} A \dot{\Rightarrow} C)$ a pour ensemble de candidats $Y \uplus Z$ et pour ensemble de solutions $Y \uplus Z$ lorsque $S \neq \varnothing$ et $T \uplus U$ lorsque $S = \varnothing$. C'est exactement pareil que $\dot{\neg} A \dot{\Rightarrow} B \dot{\vee} C$.

On a donc un candidat évident dans $(\dot{\neg} A \dot{\Rightarrow} B \dot{\vee} C) \dot{\Rightarrow} (\dot{\neg} A \dot{\Rightarrow} B) \dot{\vee} (\dot{\neg} A \dot{\Rightarrow} C)$, c'est l'identité (une fois faites nos identifications des candidats de $\dot{\neg} A \dot{\Rightarrow} B \dot{\vee} C$ comme $(\dot{\neg} A \dot{\Rightarrow} B) \dot{\vee} (\dot{\neg} A \dot{\Rightarrow} C)$ avec $Y \uplus Z$), et par la description faite des solutions, ce candidat est toujours solution. C'est précisément ce qui montre que $(\neg A \Rightarrow B \vee C) \Rightarrow (\neg A \Rightarrow B) \vee (\neg A \Rightarrow C)$ est valide dans la sémantique des problèmes finis de Medvedev. ✓

Exercice 4.5. ($\star$)

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

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

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

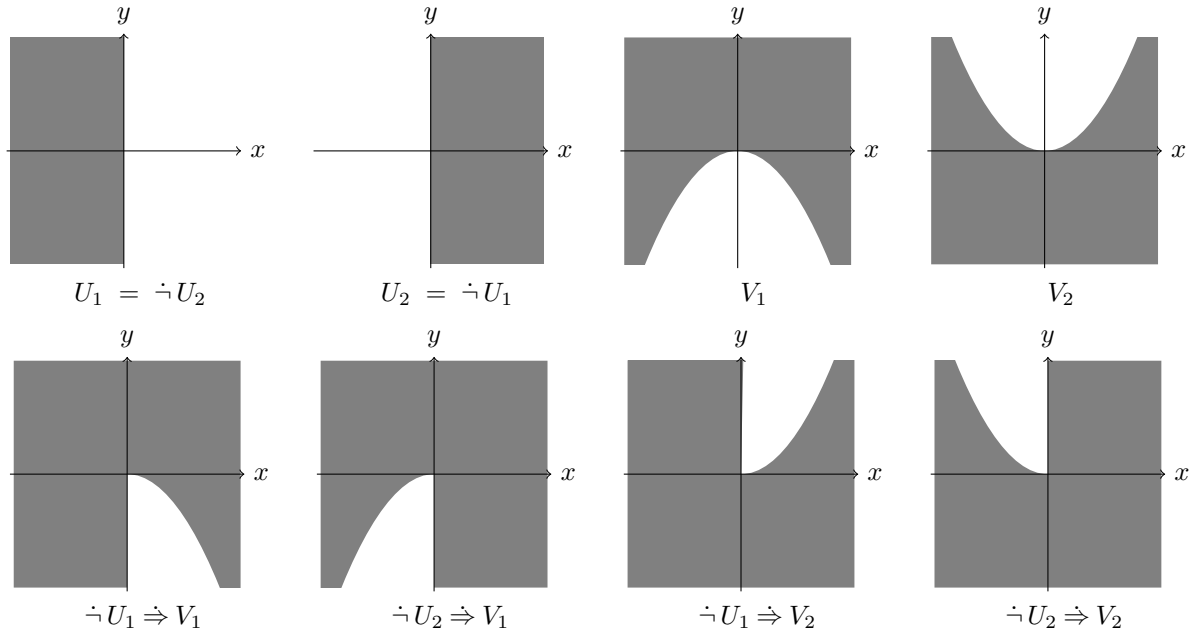
$$(\neg(A_1 \wedge A_2) \wedge (\neg A_1 \Rightarrow (B_1 \vee B_2)) \wedge (\neg A_2 \Rightarrow (B_1 \vee B_2)))$$
$$\Rightarrow ((\neg A_1 \Rightarrow B_1) \vee (\neg A_2 \Rightarrow B_1) \vee (\neg A_1 \Rightarrow B_2) \vee (\neg A_2 \Rightarrow B_2))$$

n'est pas démontrable en calcul propositionnel intuitionniste.

Corrigé. On va interpréter la formule de Tseitin dans la sémantique des ouverts de $\mathbb{R}^2$ en prenant U_1, U_2, V_1, V_2 pour A_1, A_2, B_1, B_2 respectivement, c'est-à-dire qu'on va calculer :

$$\begin{aligned}&(\dot\neg(U_1 \mathbin{\dot\wedge} U_2) \mathbin{\dot\wedge} (\dot\neg U_1 \dot\Rightarrow (V_1 \mathbin{\dot\vee} V_2)) \mathbin{\dot\wedge} (\dot\neg U_2 \dot\Rightarrow (V_1 \mathbin{\dot\vee} V_2)))\\ \dot\Rightarrow\ &((\dot\neg U_1 \dot\Rightarrow V_1) \mathbin{\dot\vee} (\dot\neg U_2 \dot\Rightarrow V_1) \mathbin{\dot\vee} (\dot\neg U_1 \dot\Rightarrow V_2) \mathbin{\dot\vee} (\dot\neg U_2 \dot\Rightarrow V_2))\end{aligned}$$

On a représenté ci-dessous en grisé les parties *ouvertes* décrites par différentes sous-expressions de cette formule :



Le membre de droite $((\dot\neg U_1 \dot\Rightarrow V_1) \mathbin{\dot\vee} (\dot\neg U_2 \dot\Rightarrow V_1) \mathbin{\dot\vee} (\dot\neg U_1 \dot\Rightarrow V_2) \mathbin{\dot\vee} (\dot\neg U_2 \dot\Rightarrow V_2))$ du $\dot\Rightarrow$ externe de la formule est la réunion des quatre dernières parties dessinées ci-dessus, c'est-à-dire le complémentaire de l'origine (noter qu'on n'attrape jamais l'origine puisqu'elle est au bord de chacune des parties dessinées, donc jamais dedans). En revanche, chacun des facteurs, $\dot\neg(U_1 \mathbin{\dot\wedge} U_2)$, $\dot\neg U_1 \dot\Rightarrow (V_1 \mathbin{\dot\vee} V_2)$ et $\dot\neg U_2 \dot\Rightarrow (V_1 \mathbin{\dot\vee} V_2)$ du membre de gauche est $\mathbb{R}^2$ tout entier, puisque $U_1 \mathbin{\dot\wedge} U_2$ est vide, et que $V_1 \mathbin{\dot\vee} V_2$ est le complémentaire de l'origine donc contient $\dot\neg U_1$ comme $\dot\neg U_2$. Finalement, pour l'expression tout entière, on trouve le complémentaire de l'origine, qui n'est pas $\mathbb{R}^2$, donc la formule de Tseitin n'est pas validée par ce choix d'ouverts, et (par *correction* de la sémantique des ouverts) elle ne peut pas être démontrable. ✓

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.)

Corrigé. **(1)** La formule $\forall A.\, (A \Rightarrow \forall C.((A \Rightarrow C) \Rightarrow C))$ est prouvée par le λ-terme $\Lambda A.\, \lambda (x : A).\, \Lambda C.\, \lambda (k : A \Rightarrow C).\, kx$. Dans l'autre sens, la formule $\forall A.\, ((\forall C.((A \Rightarrow C) \Rightarrow C)) \Rightarrow A)$ est prouvée par le λ-terme $\Lambda A.\, \lambda (h : \forall C.((A \Rightarrow C) \Rightarrow C)).\, hA(\lambda (x : A).x)$.

Si on voit ces termes comme des programmes, le premier qu'on vient de décrire prend un type A et une valeur x de ce type et convertit cette valeur en un « passage par continuation » (ou « cachée dans une clôture »), c'est-à-dire en une fonction qui prend un type C et une fonction k (qu'il faut imaginer comme une sorte continuation) et passe x comme argument à k. Le second fait la conversion inverse : il prend un type A et une valeur h passée par continuation et extrait la valeur en appliquant h au type A lui-même et à la continuation-de-fait donnée par la fonction identité.

(2) La formule $\forall A.\, \forall B.\, (A \wedge B \Rightarrow \forall C.((A \Rightarrow B \Rightarrow C) \Rightarrow C))$ est prouvée par $\Lambda A.\, \Lambda B.\, \lambda (z : A \wedge B).\, \Lambda C.\, \lambda (k : A \Rightarrow B \Rightarrow C).\, k(\pi_1 z)(\pi_2 z)$. Dans l'autre sens, la formule $\forall A.\, \forall B.\, ((\forall C.((A \Rightarrow B \Rightarrow C) \Rightarrow C)) \Rightarrow A \wedge B)$ est prouvée par $\Lambda A.\, \Lambda B.\, \lambda (h : \forall C.((A \Rightarrow B \Rightarrow C) \Rightarrow C)).\, h(A \wedge B)(\lambda (x : A).\lambda (y : B).\langle x, y \rangle)$.

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).

Si on voit ces termes comme des programmes, le premier prend en entrée deux types et un couple dont les coordonnées ont ces deux types, et le transforme en la fonction qui attend une fonction de deux arguments et l'appelle avec les deux coordonnées du couple comme arguments successifs : c'est la fonction notée `fromnative` dans la correction de l'exercice 2.4 (i.e., elle prend un couple « natif » et le convertit en sa représentation sous forme de clôture). Le second programme fait la conversion inverse et correspond à la fonction notée `tonative` dans la correction de l'exercice 2.4. La subtilité évoquée dans la correction dudit exercice revient à observer que le typage de OCaml n'est pas aussi riche que le système F.

(3) L'idée est que pour passer un type somme $P \vee Q$ par continuation on va recevoir deux fonctions, l'une invoquée dans le cas P et l'autre dans le cas Q. On peut aussi prendre son inspiration d'un des axiomes du système de Hilbert $(A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow A \vee B \Rightarrow C$. Bref, on va représenter $P \vee Q$ par $\forall C.\,(P \Rightarrow C) \Rightarrow (Q \Rightarrow C) \Rightarrow C$.

La formule $\forall A.\,\forall B.\,(A \vee B \Rightarrow \forall C.((A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow C))$ est prouvée par $\Lambda A.\,\Lambda B.\,\lambda(z : A \vee B).\,\Lambda C.\,\lambda(k : A \Rightarrow C).\,\lambda(\ell : B \Rightarrow C).\,(\texttt{match } z \texttt{ with } \iota_1 x \mapsto kx,\ \iota_2 y \mapsto \ell y)$. Dans l'autre sens, la formule $\forall A.\,\forall B.\,((\forall C.((A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow C)) \Rightarrow A \vee B)$ est prouvée par $\Lambda A.\,\Lambda B.\,\lambda(h : \forall C.((A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow C)).\,h(A \vee B)(\lambda(x : A).\iota_1^{(A,B)}x)(\lambda(y : B).\iota_2^{(A,B)}y)$.

(4) Enfin, la quantification existentielle $\exists T.\,Q(T)$ va être représentée par $\forall C.\,((\forall U.(Q(U) \Rightarrow C)) \Rightarrow C)$. La formule $(\exists T.Q(T)) \Rightarrow \forall C.((\forall U.(Q(U) \Rightarrow C)) \Rightarrow C)$ est prouvée par $\lambda(z : \exists T.Q(T)).\,\Lambda C.\,\lambda(k : \forall U.(Q(U) \Rightarrow C)).\,(\texttt{match } z \texttt{ with } \langle T, t\rangle \mapsto kTt)$. Dans l'autre sens, la formule $(\forall C.((\forall U.(Q(U) \Rightarrow C)) \Rightarrow C)) \Rightarrow (\exists T.Q(T))$ est prouvée par $\lambda(h : \forall C.((\forall U.(Q(U) \Rightarrow C)) \Rightarrow C)).\,h(\exists T.Q(T))(\Lambda U.\lambda(u : U).\langle U, u\rangle)$.

Remarque : grâce à ces manipulations, on peut réécrire tout terme du système F pour ne faire intervenir que $\Rightarrow$ et $\forall$ (il est facile de voir que $\bot$ se réécrit comme $\forall C.C$, et $\top$ comme $\forall C.(C \Rightarrow C)$). On peut même montrer que le système F limité à ces seuls connecteurs est équivalent à celui que nous avons défini, et c'est la définition la plus habituelle de ce lui-ci. Le système ainsi obtenu est plus économique et plus élégant car il ne comporte comme seule constructions des termes que l'application et les deux niveaux d'abstraction, λ et Λ.

Remarque 2 : les règles de Coq incorporent notamment celles du système F (même Fω). Par exemple, les termes que nous avons construits à la question (2) peuvent se décrire ainsi :

```
Theorem fromnative : forall (A B:Prop), (A/\B) -> forall (C:Prop),
((A->B->C)->C).
Proof. intros A B z C k. destruct z as [x y]. exact (k x y). Qed.
Theorem tonative : forall (A B:Prop), (forall (C:Prop), ((A->B->C)->C)) -> A/\B.
Proof. intros A B h. apply (h (A/\B)). intros x y. split. exact x. exact y. Qed.
```

Les λ-termes s'écrivent alors, en syntaxe Coq : `fun (A B : Prop) (z : A /\ B) (C : Prop) (k : A -> B -> C) => match z with conj x y => k x y end` pour le premier et `fun (A B : Prop) (h : forall C : Prop, (A -> B -> C) -> C) => h (A /\ B) (fun (x : A) (y : B) => conj x y)` pour le second, ce qui correspond bien à ce qui a été proposé ci-dessus. ✓

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.

Corrigé. **(1) (a)** $\lambda(h : \forall x.A(x) \Rightarrow B(x)).\,\lambda(u : \forall x.A(x)).\,\lambda(x : I).\,hx(ux)$ **(b)** $\lambda(h : \forall x.A(x) \Rightarrow B(x)).\,\lambda(v : \exists x.A(x)).\,(\texttt{match } v \texttt{ with } \langle z, w\rangle \mapsto \langle z, hzw\rangle)$ **(c)** $\lambda(p : \exists x.A(x) \Rightarrow B(x)).\,\lambda(u : \forall x.A(x)).\,(\texttt{match } p \texttt{ with } \langle z, q\rangle \mapsto \langle z, q(uz)\rangle)$

(2) On demande à OCaml de typer les trois expressions `fun h -> fun u -> fun x -> h x (u x)` et `fun h -> fun v -> match v with (z,w) -> (z, h z w)` et `fun p -> fun u -> match p with (z,q) -> (z, q(u z))` : les réponses, réécrites avec des variables et des lettres collant mieux avec ce qu'on a trouvé ci-desus, sont : $(I \to A \to B) \to (I \to A) \to (I \to B)$ pour le premier, $(I \to A \to B) \to (I \times A) \to (I \times B)$ pour le second, et $I \times (A \to B) \to (I \to A) \to (I \times C)$ pour le troisième. Ces types sont, forcément, moins précis que les types $(\prod_{x:I} A(x) \to B(x)) \to (\prod_{x:I} A(x)) \to (\prod_{x:I} B(x))$ et $(\prod_{x:I} A(x) \to B(x)) \to (\sum_{x:I} A(x)) \to (\sum_{x:I} B(x))$

et $(\sum_{x:I} A(x) \to B(x)) \to (\prod_{x:I} A(x)) \to (\sum_{x:I} B(x))$ des types que nous avons écrits à la question (1), puisque OCaml ne permet pas les types dépendants.

(On peut aussi remarquer au passage que le premier terme $\lambda hux.hx(ux)$, détypé, est le même que le combinateur S : le combinateur S est l'axiome permettant d'appliquer le *modus ponens* sous une implication, de même que notre premier terme permet d'applique le *modus ponens* sous un quantificateur universel.) ✓

6 Arithmétique du premier ordre et théorème de Gödel

Exercice 6.1. ($\star$)

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

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

(1)	$(0 + 0 = 0) \Rightarrow$ $(\forall n.((0 + n = n) \Rightarrow (0 + Sn = Sn))) \Rightarrow$ $(\forall n.(0 + n = n))$	Instance du schéma de récurrence
(2)	$\forall m.(m + 0 = m)$	Un des axiomes de Peano
(3)	$0 + 0 = 0$	∀Élim sur (2) et 0
(4)	$(\forall n.((0 + n = n) \Rightarrow (0 + Sn = Sn))) \Rightarrow$ $(\forall n.(0 + n = n))$	⇒Élim sur (1) et (3)
(5)	$\boxed{n}$	
(6)	$\quad\boxed{0 + n = n}$	
(7)	$\quad\forall m.\forall m'.((m = m') \Rightarrow$ $\quad(0 + Sn = Sm) \Rightarrow (0 + Sn = Sm'))$	Instance du schéma de substitution de l'égalité
(8)	$\quad\forall m'.((0 + n = m') \Rightarrow$ $\quad(0 + Sn = S(0 + n)) \Rightarrow (0 + Sn = Sm'))$	∀Élim sur (7) et $0 + n$
(9)	$\quad(0 + n = n) \Rightarrow$ $\quad(0 + Sn = S(0 + n)) \Rightarrow (0 + Sn = Sn)$	∀Élim sur (8) et n
(10)	$\quad(0 + Sn = S(0 + n)) \Rightarrow (0 + Sn = Sn)$	⇒Élim sur (9) et (6)
(11)	$\quad\forall m.\forall m'.(m + Sm' = S(m + m'))$	Un des axiomes de Peano
(12)	$\quad\forall m'.(0 + Sm' = S(0 + m'))$	∀Élim sur (11) et 0
(13)	$\quad0 + Sn = S(0 + n)$	∀Élim sur (12) et n
(14)	$\quad0 + Sn = Sn$	⇒Élim sur (10) et (13)
(15)	$(0 + n = n) \Rightarrow (0 + Sn = Sn)$	⇒Int de (6) dans (14)
(16)	$\forall n.((0 + n = n) \Rightarrow (0 + Sn = Sn))$	∀Int de (5) dans (15)
(17)	$\forall n.(0 + n = n)$	⇒Élim sur (4) et (16)

(On comprend pourquoi on écrit rarement de telles choses complètement.) Le λ-terme correspondant est le suivant : $\mathsf{recurr}^{(\lambda k.(0+k=k))}\,(\mathsf{defplusz}\,0)\,(\lambda(n : \mathsf{nat}).\,\lambda(h : 0 + n = n).\,\mathsf{subst}^{(\lambda k.(0+Sn=Sk))}(0 + n)\,n\,h\,(\mathsf{defplusn}\,0\,n))$ où on a

donné les noms aux axiomes defplusz : $\forall m.(m+0=m)$ et defplusn : $\forall m.\forall m'.(m+Sm'=S(m+m'))$ et pour les instances de schémas $\mathsf{recurr}^{(\lambda k.P(k))}$ pour $P(0)\Rightarrow(\forall n.(P(n)\Rightarrow P(Sn)))\Rightarrow(\forall n.P(n))$ et $\mathsf{subst}^{(\lambda k.P(k))}$ pour $\forall m.\forall n.((m=n)\Rightarrow P(m)\Rightarrow P(n))$ (le λ en exposant est ici complètement conventionnel), et enfin nat pour marquer le type des individus. Si tant est que ce programme fasse quoi que ce soit, c'est une boucle prenant n fois le successeur de 0 jusqu'à obtenir un témoignage d'égalité de $0+n$ à n. ✓

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.

Corrigé. **(1)** Le programme g_A est, comme d'habitude, bien défini grâce à l'astuce de Quine : il s'agit de construire le programme $h(e)$ qui cherche une preuve dans T de l'énoncé « si e termine, alors A » et, s'il en trouve une, termine immédiatement, et de lui appliquer le théorème de récursion de Kleene. Or l'énoncé entre guillemets peut être formalisé de façon algorithmique, et la contruction qui suit, notamment la recherche de démonstrations dans T, est bien algorithmique, donc $e\mapsto h(e)$ est bien calculable, et le théorème de récursion de Kleene s'applique. On n'a pas besoin de tester A, juste de savoir algorithmiquement reconnaître une démonstration de « si e termine, alors A ».

(2) Si g_A termine, c'est qu'il a trouvé une démonstration dans T de « si g_A termine, alors A ». Mais le fait que g_A termine implique qu'il y a une démonstration dans T de « g_A termine » (en recopiant pas à pas la trace d'exécution, ou l'arbre de calcul, de g_A). Comme T permet le *modus ponens*, on en déduit qu'il y a une démonstration dans T de A.

(3) Tout ce qui a été utilisé dans la démonstration de (2) est de l'arithmétique élémentaire, et par ailleurs valable en logique intuitionniste (aucun raisonnement par l'absurde n'a été tenu), donc l'arithmétique de Heyting comme tous les systèmes proposés dans l'énoncé prouve le point du (2). Bref, T prouve « si g_A termine, alors T prouve A ».

(4) On vient de voir au (3) que T prouve « si g_A termine, alors T prouve A » ; par conséquent, s'il prouve « si T prouve A, alors A », par *modus ponens*, il prouve « si g_A termine, alors A ». Mais c'est exactement la preuve que g_A cherche, donc g_A termine.

(5) On a vu au (2) que si g_A termine alors T prouve A, et au (4) que si T prouve « si T prouve A, alors A » alors g_A termine. On en déduit si T prouve « si T prouve A, alors A » alors T prouve A. C'est le théorème de Löb. ✓

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.

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.

(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.

Corrigé. **(1)** Comme dans le (1) de l'exercice 6.2, le programme est bien défini par l'astuce de Quine et par le fait que vérifier une preuve dans T est algorithmique.

(2) Par construction de h, si T prouve que h termine alors h finira par trouver cette preuve, et terminera. (*Attention :* ici il ne faut pas dire « si T prouve que h termine alors h termine puisqu'on l'a prouvé » : on n'a pas fait l'hypothèse que T dit la vérité sur les arrêts de machines de Turing.)

(3) Tout ce qui a été utilisé dans la démonstration de (2) est de l'arithmétique élémentaire, et par ailleurs valable en logique intuitionniste (aucun raisonnement par l'absurde n'a été tenu), donc l'arithmétique de Heyting comme tous les systèmes proposés dans l'énoncé prouve le point du (2). Bref, T prouve « si T prouve que h termine, alors h termine ».

(4) On a montré question (3) que T prouve « si T prouve que h termine, alors h termine ». Mais on a montré à l'exercice 6.2 que si T prouve « si T prouve que h termine, alors h termine », alors T prouve que h termine (c'est le théorème de Löb avec pour A l'énoncé « h termine »). Par conséquent, T prouve que h termine. Donc, par la question (1), h termine effectivement. ✓

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 ?

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 $10^{10^{1000}}$ itérations de la fonction $n \mapsto A(n,n,n)$, où A est la fonction d'Ackermann, en partant de $10^{10^{1000}}$ (*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.

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

Corrigé. **(1)** Le programme p n'est pas très long à écrire : il s'agit de calculer $10^{10^{100}}$, ce qui n'est pas difficile, puis d'effectuer une boucle finie sur tous les entiers pouvant coder une chaîne de longueur au plus $10^{10^{100}}$, de tester si chacun est une démonstration valable dans PA (ce qui est algorithmique et pas terriblement difficile) avec pour conclusion $\varphi_e(i)\downarrow$ et, le cas échéant, d'utiliser l'interpréteur universel pour simuler l'exécution de $\varphi_e(i)$ et l'ajouter à N, puis finalement renvoyer N. Aucune de ces étapes n'est très difficile, et on peut certainement dire que le programme est de longueur bien inférieure à 10^{100}, par exemple.

Quant à sa terminaison, la seule partie qui n'est pas évidente est le fait que $\varphi_e(i)\downarrow$. On en a une preuve dans PA (par construction du programme !), mais encore faut-il que PA dise la vérité, ce qui constitue précisément l'hypothèse $\Sigma_1\mathsf{Sound}(\mathsf{PA})$ introduite par l'énoncé.

(2) Une fois admise l'hypothèes $\Sigma_1\mathsf{Sound}(\mathsf{PA})$, il est évident que p termine, puisqu'il exécute $\varphi_e(i)$ une fois qu'il a une preuve dans PA que $\varphi_e(i)\downarrow$, donc toutes ces exécutions terminent bien par $\Sigma_1\mathsf{Sound}(\mathsf{PA})$. Tout le reste est une boucle finie, donc termine certainement.

Comme on a vu en cours que si un programme s'arrête, ce fait est prouvable dans PA (en transformant une trace d'exécution complète en démonstration), c'est le cas pour notre programme p.

(3) Soit q le programme qui implémente la fonction d'Ackermann et s'en sert pour calculer $10^{10^{1000}}$ itérations de la fonction $n \mapsto A(n,n,n)$ en partant de $10^{10^{1000}}$: ce programme termine car la fonction d'Ackermann est bien définie. Le *fait* que ce programme termine est facile à prouver, et cette preuve, qui repose sur des considérations arithmétiques (des récurrences imbriquées) est menable dans PA ; de plus, elle n'est pas terriblement longue, certainement bien plus courte que 10^{100}. Par conséquent, p va, entre autres, tomber sur cette preuve au cours de son énumération, donc il va exécuter p, donc ajouter son résultat à N. Donc la valeur de retour de p est supérieure à celle de q, et même considérablement plus grand puisque q va ajouter le résultat de tous les calculs de ce genre dont on peut prouver la terminaison en moins de $10^{10^{100}}$ symboles.

Au cours de l'exécution de p, on doit parcourir toutes les démonstrations de longueur au plus $10^{10^{100}}$: cette énumération est certes très longue (en gros une exponentielle de plus), mais complètement minuscule par rapport au temps d'exécution des programmes car on vient de voir qu'il y en a qui calculent des nombres considérablement plus gigantesques (et parmi les programmes il y en a qui calculent ce nombre puis attendent ce nombre d'étapes).

(4) On a vu en (2) que PA prouve que p termine, mais il n'est pas possible qu'il le prouve en moins de $10^{10^{100}}$ symboles, car si tel était le cas, p ferait partie des programmes exécutés lors de la boucle de p, donc le résultat ferait partie de ceux ajoutés à N, donc on aurait $N > N$.

(5) Le fait que p termine est prouvable dans ZFC grâce au raisonnement expliqué en (2), et la preuve n'est pas très longue, certainement bien plus courte que 10^{100} symboles, même quand on ajoute la preuve de $\Sigma_1\mathsf{Sound}(\mathsf{PA})$ qui n'est elle-même pas bien longue d'après l'énoncé. Comme on vient de voir que la plus courte preuve de la terminaison de p dans PA est, pour sa part, plus longue que $10^{10^{100}}$ symboles, la preuve dans ZFC est beaucoup plus courte. ✓

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).)

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).

Corrigé. Traitons d'abord la question subsidiaire. Si les tableaux sont les suivants :

$$\begin{array}{lll} p_A[A] = B & p_A[B] = B & p_A[C] = C \\ p_B[A] = A & p_B[B] = C & p_B[C] = C \\ p_C[A] = A & p_C[B] = B & p_C[C] = A \end{array}$$

alors chacun des tableaux vérifie les deux contraintes (il renvoie toujours l'étiquette d'une des deux autres portes, et si on le consulte sur l'étiquette d'une des deux autres portes, il renvoie celle-ci) ; or ces tableaux sont complètement symétriques (ils sont invariants par les permutations cycliques de A, B, C) donc ils ne peuvent pas permettre de déduire l'information de l'emplacement du dragon qui pourrait être derrière n'importe quelle porte.

Il s'agit donc de faire quelque chose avec les programmes qu'on ne peut pas faire avec de simples tableaux. Ceci suggère d'utiliser le théorème de récursion de Kleene.

En s'inspirant de l'exercice 1.16(1), si Z est une des trois portes et X, Y les deux autres, on va définir le programme q_Z suivant : il invoque le programme p_Z sur q_Z lui-même et ensuite, si p_Z termine et renvoie X ou Y, il échange X et Y (autrement dit, si p_Z appelé sur q_Z renvoie X, alors il renvoie Y, et si p_Z appelé sur q_Z renvoie Y, alors il renvoie X), tandis que dans tout autre cas il fait une boucle infinie. La construction de ce q_Z, et notamment le fait que q_Z fasse appel à lui-même dans sa définition, est justifiée par l'astuce de Quine. Si Z est la porte au dragon, alors, d'après les garanties qu'on a reçues, p_Z invoqué sur q_Z va forcément terminer et renvoyer une des étiquettes X, Y (les deux sont sûres). Si Z est une porte sûre, mettons pour fixer les idées que X soit la porte au dragon : alors p_Z invoqué sur q_Z ne peut pas renvoyer X, car si tel était le cas, q_Z renverrait Y (puisqu'il échange les deux réponses X, Y), et par les garanties qu'on a reçues, p_Z invoqué sur q_Z doit renvoyer Y, contradiction.

Maintenant, on lance en parallèle p_A sur q_A, p_B sur q_B, et p_C sur q_C. D'après ce qui a été dit au paragraphe précédent, aucun des trois ne peut renvoyer l'étiquette de la porte du dragon, et l'un des trois (celui de la porte du dragon) devra finir par renvoyer l'étiquette d'une porte sûre. Il suffit donc d'attendre que l'un des trois termine et renvoie une étiquette dans $\{A, B, C\}$, et, quand ça se produit, ouvrir la porte en question. ✓

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

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

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

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

(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.)

Corrigé. Rappelons que, pour la réalisabilité propositionnelle, $\dot\neg P$ vaut $\mathbb{N}$ lorsque P est vide, et $\varnothing$ lorsque P n'est pas vide (et symétriquement, $\dot\neg\dot\neg P$ vaut $\varnothing$ lorsque P est vide, et $\mathbb{N}$ lorsque P n'est pas vide).

Montrons que la formule est réalisable : pour ça, on va chercher à en produire un réalisateur r (qui doit être le même quelles que soient $A, B, C \subseteq \mathbb{N}$). Ce réalisateur est un programme prenant six entrées. Les trois premiers sont des réalisateurs de $(\dot\neg\dot\neg A \mathrel{\dot\Rightarrow} \dot\neg B \mathbin{\dot\wedge} \dot\neg C)$ et symétriquement, et ne nous intéresseront pas pour leurs valeurs, ce sont simplement des promesses que si A est non-vide alors B et C sont vides et symétriquement : on retient simplement de ces trois entrées la garantie qu'exactement une des trois parties A, B, C est non-vide (« a un dragon »). On va appeler p_A, p_B, p_C les trois autres entrées (quitte à les modifier un tout petit peu comme on va l'expliquer). Par exemple, p_A doit réaliser $((\dot\neg A \mathrel{\dot\Rightarrow} (\dot\neg B \mathbin{\dot\vee} \dot\neg C)) \mathrel{\dot\Rightarrow} (\dot\neg B \mathbin{\dot\vee} \dot\neg C)$, c'est-à-dire que si on lui passe en argument un réalisateur q de $\dot\neg A \mathrel{\dot\Rightarrow} (\dot\neg B \mathbin{\dot\vee} \dot\neg C)$ il doit terminer et renvoyer un réalisateur de $\dot\neg B \mathbin{\dot\vee} \dot\neg C$; or un réalisateur de $\dot\neg B \mathbin{\dot\vee} \dot\neg C$ est la donnée d'un élément de $\dot\neg B$ ou de $\dot\neg C$ ainsi que l'information duquel on a pris : mais cela revient simplement à donner l'étiquette d'un des deux ensembles B ou C qui est vide (« n'a pas de dragon »). Donc la contrainte sur q est que si A est vide il renvoie l'étiquette d'un des deux ensembles B ou C qui est vide (et si A n'est pas vide, n'y a aucune contrainte sur q) ; et la contrainte sur p_A est que si on lui passe un tel q, il renvoie lui-même l'étiquette d'un des deux ensembles B ou C qui est vide. Et au final, notre programme r doit renvoyer l'étiquette d'un des trois ensembles A, B, C qui est vide.

Ce sont donc exactement les termes de l'énigme de l'exercice 7.1 : le réalisateur r recherché pour la formule est le programme qui prend p_A, p_B, p_C comme on l'a dit, qui effectue le calcul décrit dans la solution de 7.1, et renvoie l'étiquette qu'il a trouvé. La formule de Plisko est bien réalisable.

Néanmoins, cette formule n'est pas Medvedev-valide : en effet, appelons A, B, C trois problèmes finis ayant chacun un candidat et dont un seul a une solution (« un dragon »). Le problème décrit par l'ensemble de la formule consiste toujours à trouver une solution de l'énigme mais cette fois en interprétant les programmes comme des tableaux finis (fonctions entre ensembles finis). Or on a expliqué dans la question subsidiaire de l'exercice 7.1 que l'énigme n'avait pas de solution dans ces conditions : c'est dire que la formule n'est pas Medvedev-valide.

(Notamment, la formule de Plisko n'est pas démontrable en calcul propositionnel intuitionniste, par *correction* de la sémantique de Medvedev.) ✓