Logique - Calculabilité - Complexité

Université de Montpellier

Cours 2024

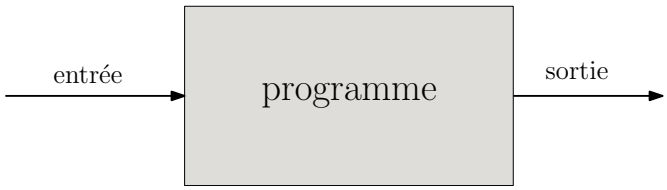
cours de Bruno Durand
rédigé avec l'aide de Mathieu Mari et Christian Rétoré

Chapter 1

Calculabilité

1.1 Notions de base

La calculabilité c'est s'intéresser à la question "qu'est-ce qu'on peut faire avec les algorithmes en général ?" et aussi "qu'est-ce qu'on ne peut pas faire?". On imagine les algorithmes un peu comme des boîtes noires avec une entrée et une sortie.



L'entrée et la sortie sont tout deux des mots sur un alphabet fini, généralement noté Σ dans ce cours. Typiquement, $\Sigma = \{0, 1\}$ (codage binaire). Un mot est une suite finie de lettres de cet alphabet. Par exemple 0010110 est un mot sur l'aphabet $\{0, 1\}$. L'ensemble des mots sur cet alphabet est noté Σ^*.

1.1.1 Encodage des entrées, des sorties et des programmes.

Pour se simplifier la vie, on fait correspondre bijectivement à chaque mot sur $\{0, 1\}$ en un entier naturel, c'est-à-dire un élément de $\mathbb{N}$. Pour trouver l'entier n associé à un mot m, on procède de la manière suivante. On commence par ajouter un 1 devant le mot, pour obtenir un nouveau mot $m' = 1m$. Puis on associe l'entier n' dont l'écriture en binaire est m'. Enfin on renvoie l'entier $n = n' - 1$.

Par exemple le mot 010 est associé au nombre 17. Il n'est pas difficile de ce convaincre ce procédé établit une bijection entre $\{0, 1\}^*$ et $\mathbb{N}$. Il n'est pas non plus difficile d'écrire un programme réalisant cette bijection ainsi que la bijection inverse (c'est-à-dire la fonction qui a un entier associe le mot correspondant).

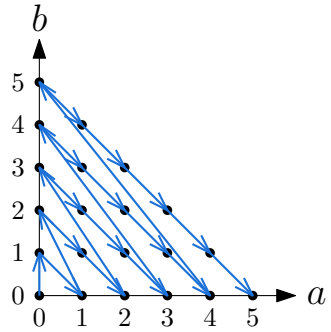
La première opération consistant à rajouter un 1 est là pour par exemple distinguer les mots 00101 et 101. Remarquer que l'encodage binaire d'un entier non nul commence toujours par un 1. L'opération -1 permet d'atteindre l'entier 0, et ainsi obtenir une fonction bijective plutôt que simplement injective.

On a ainsi contruit un correspondance *bijective* entre les mots de d'un alphabet avec deux symboles et les entiers naturel. Une telle bijection existe aussi pour les alphabets de taille 3, 4, etc. Dans la suite du cours on va donc considérer que tout entrée et toute sortie est un entier naturel, c'est à dire un élément de $\mathbb{N}$.

Codage d'une paire d'entier. Comment faire si notre programme prend en entrée plusieurs paramètres? par exemple 2? Pour cela, on va coder chaque paire d'entier par un unique entier, et cela de façon bijective. On notera l'entier codant la paire a, b par $\langle a, b\rangle_2$ (avec des crochets).

Attention, ici on code des couples (a, b) et non des paires $\{a, b\}$ mais le terme paire est consacré par l'usage. Ainsi, si $a \neq b$ alors $\langle a, b\rangle \neq \langle b, a\rangle$. De plus, quand il n'y a pas d'ambiguité on préfèrera écrire $\langle a, b\rangle$ plutot que $\langle a, b\rangle_2$ car c'est plus clair.

On cherche donc à construire une fonction $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ qui est bijective. Pour celà il suffit d'ordonner l'ensemble des couples comme sur le schéma suivant et puis leur associer un numéro (un entier) correspondant à cet ordre.



Par exemple: $\langle 0, 0\rangle_2 = 0$, $\langle 2, 0\rangle_2 = 5$, $\langle 1, 3\rangle_2 = 11$.

On peut se convaincre facilement qu'il existe un programme qui calcule cet encodage.

A partir du codage $\langle a, b\rangle$ (en un entier) d'un couple, on peut retrouver facilement les deux entiers correspondant grâce aux *projections*. On note π_1 la première projection et π_2 la deuxième. Ce sont des fonctions de N dans N telles que pour tout entier n, on a $\langle \pi_1(n), \pi_2(n)\rangle_2 = n$. Encore une fois, il existe des programmes qui calculent chacune des projections.

Encodage des tuples. Mais que faire si notre programme à 3 entrées? 4 ? 1000? Pour coder un t-uplet $(a_1, \ldots, a_t)$ de façon bijective par un entier $\langle a_1, \ldots, a_t\rangle_t$, on procède de façon inductive. On suppose qu'on a déjà construit la fonction d'encodage $\langle \cdot, \ldots, \cdot\rangle_{t-1} : \mathbb{N}^{t-1} \to \mathbb{N}$ d'un $t-1$-uplet, et on définit

$$\langle a_1, \ldots, a_t\rangle_t = \langle \langle a_1, \ldots, a_{t-1}\rangle_{t-1}, a_n\rangle_2$$

Autrement dit, simplement grâce à la fonction d'encodage des couples, on peut construire les fonctions d'encodage de tuples de n'importe quelle arité. Grâce à ces fonction d'encodage on peut faire l'hypothèse dans dans ce cours l'entrée de nos programmes sera simplement un entier, et ceci sans perte de généralités.

Un petit problème ici est que le programme doit connaître son nombre de paramètre t afin de décoder correctement son entrée. Pour éviter ce problème, on peut utiliser une autre structure de données : *les cortèges*. On va alors coder l'arité t au début de chaque entrée. Par

exemple, pour encoder le 3-uplet $(54, 7, 12)$ on utilisera l'entier $\langle 3, \langle 54, 7, 12\rangle\rangle$. Pour simplifier les notations, on se passe souvent d'écrire l'arité en indice et ici il faut comprendre $\langle 3, \langle 54, 7, 12\rangle_3\rangle_2$.

Codage des programmes. Les programmes peuvent être décrits par une suite finie de symboles d'un alphabet fini, et on peut donc, comme les entrées et les sorties, les coder par des entiers. Dans tout ce cours on considère donc qu'on a fixé une façon de coder les programmes par des entiers. On peut considérer que les programmes ont chacun un numéro et qu'à partir du numéro, on est capable de retrouver le code du programme. Attention: le codage est uniquement calculé à partir du texte du programme et non la fonction que le programme réalise. Dans la suite on va associer un programme avec son numéro.

1.1.2 Les programmes

Pour un programme a, on note $[a|x]$ le fait d'exécuter le programme (codé par l'entier) a sur l'entrée x. Lorsqu'on exécute ce programme a sur l'entrée x, 3 choses peuvent se passer:

- Le programme termine au bout d'un certain temps (fini) et retourne la valeur y (que l'on suppose être un entier sans perte de généralités). Dans ce cas, on dira que le programme p *converge*, *termine*, ou s'*arrête* sur l'entrée x; On note cela $[a|x] = y$ ou, si on s'intéresse juste à dire que le programme s'est arrêté, on pourra juste noter $[a|x]\downarrow$.

- Le programme plante ou bien s'arrête en renvoyant une erreur (division par zero, segmentation fault, fichier à importer qui n'existe pas, ...). Dans ce cas on dira que le programme p *plante* sur l'entrée x.

  On considèrera de même un programme qui boucle à l'infini (par exemple un boucle `while true`). Dans ce cas on dira que le programme p *diverge* sur l'entrée x. On note le plantage et la divergence de la même façon $[a|x]\downarrow$.

Remarque : en calculabilité, on ne s'intéresse pas au temps que prend le programme pour s'arrêter mais seulement le fait qu'il s'arrête ou pas.

Attention, si on écrit $[a|x] = y$ on veut dire que le proramme a sur l'entrée x s'arrête et que, en plus de s'arrêter, il renvoie y.

1.1.3 Les fonctions

Dans le cadre de ce cours, une fonction est un opérateur qui associe à tout entier d'une partie E de $\mathbb{N}$, un élément de $\mathbb{N}$, c'est-à-dire un entier. Les fonctions sont donc *partielles* par défaut. On appelle le sous-ensemble E le *domaine de définition* (ou simplement le *domaine*) de la fonction. Le domaine d'une fonction f est notée $dom(f)$.

Une fonction est *totale* si son domaine de définition est $\mathbb{N}$ tout entier, c'est-à-dire si $domf = \mathbb{N}$.

L'*image* d'une fonction f, notée $Im(f)$ est défini par $Im(f) = \{y, \exists x, f(x) = y\}$. C'est l'ensemble des entiers qu'on peut trouver à la sortie de f.

Dans ce cours, on va s'intéresser aux fonctions que peuvent calculer les programmes. Ainsi, la fonction calculée par le programme a est notée $[a|\cdot]$. Le domaine $dom[a|\cdot]$ de cette fonction correspond exactement à l'ensemble des entrées sur lesquelles le programme a converge (donc donne un résultat). On l'écrit aussi W_a. Si on veut écrire qu'une entrée x n'est pas dans le domaine de définition d'une fonction f, on pourra écrire $f(x) = \perp$.

Une fonction particulière, aisément programmable, est la fonction définie nulle part (et dont l'image est donc vide). On la note $f(.) = \perp\!\!\!\perp$.

1.1.4 Les ensembles

Tous les ensembles considérés dans ce cours sont des sous-ensembles de $\mathbb{N}$. La *fonction caractéristique* d'un ensemble $E \subseteq \mathbb{N}$, notée χ_E, est définie par $\chi_E(x) = 1$ si $x \in E$ et $\chi_E(x) = 0$ sinon. Remarquer que la fonction caractéristique d'un ensemble est une fonction totale. Mais attention à bien regarder χ_E comme une fonction et non pas comme un programme.

Le complémentaire de l'ensemble E est noté $\bar{E} = \mathbb{N} \setminus E$ et contient les entiers qui ne sont pas dans E.

1.2 Ensembles récursifs et énumérables

1.2.1 Ensembles récursifs

Définition 1.2.1. Une fonction f est *calculable*, ou *récursive*, s'il existe un programme qui la calcule, c'est-à-dire, s'il existe un programme a tel que pour tout x

$$x \in dom(f) \Longleftrightarrow [a|x] \downarrow \text{ et, dans ce cas, } [a|x] = f(x) \ .$$

Exemples : la fonction qui à tout entier n associe 0 si n est pair et 1 sinon, est calculable. La fonction définie nulle part $\perp\!\!\!\perp$ est calculable. La fonction identité sur $\mathbb{N}$ est calculable.

Remarque : Une fonction est calculable totale, ou récursive totale, si non seulement il existe un programme qui la calcule, mais aussi si son domaine de définition est $\mathbb{N}$ tout entier.

Proposition 1.2.1. *Il existe des fonctions non calculable.*

Pour démontrer cet énoncé, on ne va pas donner à ce stade un exemple d'une telle fonction, mais seulement montrer qu'il en existe. Rassurez-vous, plus tard, on donnera des exemples explicites.

Preuve. Il y a trop de fonctions, et pas assez de fonctions calculables, donc il existe nécessairement des fonctions qui ne sont pas calculables (même une infinité). En effet, l'ensemble des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ est un ensemble non dénombrable, alors que l'ensemble des programmes est dénombrable (on a vu qu'on pouvait associer à chaque programme un entier différent). Or, il y a moins de fonctions calculables que de programmes (puisque plusieurs programmes différents peuvent correspondre à la même fonction). Donc l'ensemble des fonctions calculables est également dénombrable. □

Définition 1.2.2. Un ensemble E est *récursif* si sa fonction caractéristique est calculable.

Une définition équivalente est qu'un ensemble est récursif si il existe un programme qui prend en entrée x et renvoie 1 si $x \in E$ et 0 sinon. En particulier, ce programme ne plante jamais et en quelque sorte nous dit si oui ou non l'entrée est dans l'ensemble E. On peut donc alors utiliser l'appartenance à E dans un test d'un programme. Notamment, on pourra écrire dans: "`if` $x \in E$ `then ...`".

Par exemple, l'ensemble des images JPEG, l'ensemble des programmes python correctement écrits, etc, sont des ensembles récursifs.

Remarque. Un ensemble est récursif si et seulement si son complémentaire l'est aussi. En effet il suffit d'échanger le 0 de la réponse par un 1 et réciproquement.

1.2.2 Les ensembles énumérables

Les ensembles énumérables ont un rôle central dans l'informatique en général, et du coup aussi dans la calculabilité.

Définition 1.2.3. Un ensemble E est *énumérable* si E est le domaine de définition d'une fonction calculable, c'est-à-dire si il existe a tel que $E = dom[a|\cdot]$ (qu'on peut aussi écrire $E = W_a$.

On peut caractériser ces ensembles de façons très différentes, ce qui va expliquer le terme "énumérable".

Théorème 1.2.2 (Théorème de caractérisation des énumérables)**.** *Les propositions suivantes sont équivalentes :*

1. *E est* énumérable *(rappel de la définition) : si* $\exists a\ E = W_a = \{x,\ [a|x]\downarrow\}$
2. *E admet une* fonction d'énumération *calculable :* $\exists b, E = Im[b|\cdot] = \{x,\ \exists y\ [b|y] = x\}$
3. *E est vide ou admet une* fonction d'énumération totale *calculable :* $\exists b, E = Im[b|\cdot] = \{x,\ \exists y\ [b|y] = x\}$

Ce théorème est démontré dans l'exercice 1 du TD1.

Une autre façon de se représenter les ensembles énumérables vient de ce théorème d'équivalence. On peut les imaginer comme les ensembles E pour lesquels il existe une machine (un programme), qui tourne (potentiellement indéfiniment) et nous affiche les éléments de E un à un, dans le désordre. On a juste la garantie que si un entier x est dans E, alors il sera affiché à un moment par la machine. Cependant, si cet ensemble énumérable n'est pas récursif, on ne peut pas tester si un tel élément x est dans E. En effet, il se pourrait qu'après avoir attendu un certain temps t, la machine n'a toujours pas affiché l'élément x, et on ne peut pas en déduire que x n'appartient pas à l'ensemble, car peut-être que la machine va afficher x en un temps plus long.

Attention à notre façon de décrire les programmes : si E est un ensemble énumérable, il est délicat d'écrire dans un programme "`if` $x \in E$ `then A else B`". En effet, si $x \in E$, alors le programme va s'en apercevoir (en utilisant par exemple le programme qui s'arrête sur E)

et donc il va exécuter les instructions du bloc `A`, mais dans le cas contraire si $x \notin E$ notre programme va diverger et n'exécutera jamais l'opération `B`. Ainsi lorsque E est énumérable, il sera prudent d'écrire "`if` $x \in E$ `then A` sans le `else`.

1.2.3 Relations entre ces types d'ensemble

Proposition 1.2.3. *Les ensembles récursifs sont énumérables.*

Preuve. Soit E un ensemble récursif. Par définition, il existe un programme a tel que pour tout x, $[e|x] = 1$ si $x \in E$ et $[e|x] = 0$ sinon. On construit le programme suivant : à x on associe 1 si $[e|x] = 1$, et sinon on associe $\perp$. Cette fonction est évidemment calculable puisque e est un programme définit partout. On vérifie que son domaine est égal à E. □

Être récursif est donc plus fort qu'être énumérable. Pour qu'un ensemble énumérable soit aussi récursif, il suffit (et il faut) que son complémentaire soit énumérable. C'est ce que dit le théorème de Post suivant :

Théorème 1.2.4 (Théorème de Post)**.** *Si E est énumérable et si son complémentaire $\bar{E}$ l'est aussi, alors E est récursif.*

Pour rappel, $\bar{E}$ désigne le complémentaire de E. On verra plus tard qu'il existe des ensembles énumérables qui ne sont pas récursifs. Et en particulier, d'après ce théorème, il existe des ensembles énumérables dont le complémentaire ne l'est pas (contrairement aux ensembles récursifs). Ce n'est pas surprenant : quand un ensemble est énumérable, on sait qu'il existe une machine qui affiche successivement ses éléments, donc si on voit un élément affiché on a la garantie que cet élément est dans l'ensemble, mais à l'inverse on ne pourra jamais avoir la certitude qu'un élément donné n'y est pas.

Pour démontrer le théorème de Post, on va utiliser la fonction `step`. Cette fonction permet de simuler l'exécution d'un programme pendant un certain temps. Elle sera particulièrement utile pour concevoir des programmes qui convergent, mais qui utilisant en routine d'autres programmes pouvant diverger sur certaines entrées.

Proposition 1.2.5. *Il existe une fonction calculable totale, noté* `step`*, qui prend en argument: un programme a, une entrée x et un temps t (donc 3 entiers codés en 1 entrée), et telle que*
• $\texttt{step}\langle a, x, t\rangle = 0$ *si l'exécution du programme a sur l'entrée x n'a pas encore convergé au temps t;*
• $\texttt{step}\langle a, x, t\rangle = y + 1$ *si cette exécution a convergé avant le temps t et a renvoyé la valeur y.*

Le $+1$ sur la sortie permet de distinguer le cas où le programme a converge vers la valeur 0. Cette fonction est bien sûr totale et dans tous nos environnements de calcul, elle est calculable, avec des notions de temps de calcul qui dépendent de l'environnement (langage, OS, etc).

On a donc $[a|x] \uparrow \Leftrightarrow \ \forall t\ \texttt{step}\langle a, x, t\rangle = 0$. Autrement dit $[a|x] \downarrow \Leftrightarrow \ \exists t\ \texttt{step}\langle a, x, t\rangle \neq 0$. On a aussi $[a|x] = y \ \Leftrightarrow \ \exists t\ \texttt{step}\langle a, x, t\rangle = y + 1$

Preuve du théorème de Post. Idée de la preuve : soit E un ensemble énumérable dont le complémentaire $\bar{E}$ l'est aussi. On sait qu'on a un programme a qui s'arrête exactement sur E (on peut écrire $E = dom[a|\cdot]$) et un autre programme $\bar{a}$ qui s'arrête exactement sur le

complémentaire de E noté $\bar{E}$. On va faire tourner les deux programmes en parallèle et observer lequel s'arrête. Si c'est a, alors l'entrée est dans E si c'est $\bar{a}$ qui s'arrête, alors l'entrée est dans $\bar{E}$.

Plus formellement, on définit donc un programme e suivant qui prend en argument x et renvoie 1 si $x \in E$ et 0 sinon. Il y a plusieurs façons de faire ce programme.

Examinons ici une première solution

$[e|x]$:

$t \leftarrow 0$, StopA $\leftarrow$ False, StopNonA $\leftarrow$ False

while StopA $\vee$ StopNonA = False

$t \leftarrow t + 1$

if $\texttt{step}\langle a, x, t\rangle \neq 0$ then StopA $\leftarrow$ True

if $\texttt{step}\langle \bar{a}, x, t\rangle \neq 0$ then StopNonA $\leftarrow$ True

if StopA then return 1 else return 0

Le programme e code la fonction caractéristique de E, ce qui prouve que E est récursif.

Expliquons un peu cette algorithmique. Dans notre pseudo langage on fait des affectations à des variables sans les déclarer, leur type étant implicite : $z \leftarrow 0$ induit donc que z est une variable entière et StopA $\leftarrow$ False induit que StopA est une variable booléenne. Nous utilisons l'indentation pour déterminer la portée des boucles.

Notre premier while signifie : *tant que ni le programme a ni le programme $\bar{a}$ ne se sont arrêtés, on incrémente le temps de calcul t.* Quand on sort de cette boucle c'est que l'une ou l'autre des variables StopA et StopNonA est non nulle. On en déduit si $x \in E$ ou si $x \notin E$. □

Examinons maintenant une autre solution. Celle-ci n'utilise pas les programmes dont le domaine de convergence est E et $\bar{E}$ mais utilise leurs programmes d'énumération notés b et $\bar{b}$.

$[e|x]$:

$z \leftarrow 0$

repeat:

$y \leftarrow \pi_1(z)$

$t \leftarrow \pi_2(z)$

if $\texttt{step}\langle b, y, t\rangle = x + 1$ then

return 1

else if $\texttt{step}\langle \bar{b}, y, t\rangle = x + 1$ then

return 0

else

$z = z + 1$

Le programme e code la fonction caractéristique de E, ce qui prouve que E est récursif.

Ici, on cherche à voir si b ou $\bar{b}$ nous donnent en sortie x pour une certaine entrée y qu'on ne connait pas à l'avance. Alors on voudrait essayer tous les y. Mais ce n'est pas possible car si un

des programmes plante sur un de ces y on va être bloqué et on ne va plus pouvoir essayer les autres valeurs pour y. Alors on fait des calculs pendant un temps limité t et on se débrouille pour essayer tous les temps et toutes les entrées. Du coup z est ici une variable qui parcourt toutes les possibilités, mais qui en réalité représente deux variables y et t.

Une autre remarque sur ce programme porte sur ce qu'on appelle un échappement. C'est à dire le fait que nous ayons mis les return dans une boucle. Ce n'est pas considéré comme très propre en algorithmique et on pourrait les placer après la boucle repeat. Mais il faut bien penser qu'en calculabilité, nous faisons de l'algorithmique spéciale car les cas de plantage d'un programme nous intéressent alors qu'en algorithmique, on souhaite à tout pris les éviter. Donc ce n'est pas très grave pour nous d'avoir un return dans une boucle.

1.3 Fondamentaux de la Calculabilité

1.3.1 L'universalité

Théorème 1.3.1 (Universalité)**.** $\exists u\ \forall x, y\ [u|\langle x, y\rangle] = [x|y]$.

Autrement dit, il existe un programme *universel* u qui prend en entrée (le numéro d') un programme x et une entrée y et renvoie en sortie la même chose que ce que donne le programme x sur l'entrée y.

On peut encore le voir comme l'existence d'un programme qui est capable de lancer l'exécution du programme x sur l'entrée y. Dans de nombreux langages de programmation, on utilise pour cela un appelle au système d'exploitation.

Ce programme universel est propre à l'environnement, au langage de programmation utilisé, etc. Par exemple il existe une machine de Turing universelle qui permet de simuler n'importe quelle machine de Turing, et la preuve de ce résultat utilise la définition des machines de Turing.

Ce théorème est donc propre à chaque modèle de calcul. En d'autres termes, on n'utilisera que des modèles de calcul ayant une machine universelle. Il y a donc un théorème d'universalité par modèle de calcul.

1.3.2 Indécidabilité

On a montré qu'il existe des fonctions non calculables (Proposition 1.2.1), mais on n'a pas explicité de telle fonction. C'est le but de cette section.

On commence par rappeler ce qu'est un problème de décision.

1.3.3 Problèmes de décision

Les problèmes de décision sont une façon équivalente mais souvent plus intuitive de définir des ensembles ou des fonctions.

Un *problème de décision* prend des entrées appelées généralement *instance* et pour chaque entrée, pose une question concernant cette entrée au auquel on répond par OUI ou NON. Les

instances *positives* sont les instances pour lesquelles la réponse à la question est OUI et les instances négatives sont les instances pour lesquelles la réponse à la question est NON.

NOM DU PROBLÈME:
Entrée: $x \in E$
Question: Est-ce que, lorsque $x \in E$, x satisfait une certaine propriété P ?

Des exemples :

PREMIER :
Entrée: x un entier
Question: Est-ce que x est un nombre premier ?

CONNEXE :
Entrée: x un graphe
Question: Est-ce que le graphe x est connexe?

Pour le problème CONNEXE, on suppose qu'on s'est fixé une méthode de codage de l'ensemble des graphes par des entiers. En d'autres termes, on a une structure de données fixée pour les graphes – deux structures sont fréquentes : les tables d'adjacences d'une part et les représentations par liste d'arêtes d'autre part.

Les instances négatives de ce problème est l'ensemble des x qui sont des graphes ayant plusieurs composantes connexes.

A chaque ensemble A, on peut définir un problème de décision associé :

APPARTENANCE$_A$:
Entrée: x un entier
Question: $x \in A$?

Dans l'autre sens, si on a un problème de décision, on obtient deux ensembles : celui des entrées valides E et l'ensemble des instances positives A. Ce dernier ensemble est donc par définition inclus dans E. Si maintenant il est facile de déterminer si les entrées sont valides, par exemple si $E = \mathbb{N}$ ou plus généralement si E est récursif, alors le problème de décision est caractérisé par l'ensemble des instances positives A. C'est ce dernier cas qui nous intéresse.

Note culturelle. Les problèmes de décision sont très utilisés en informatique pour analyser la difficulté des questions qu'on aimerait faire résoudre à des informaticiens. En calculabilité on s'intéresse aux problèmes qu'on peut résoudre par des algorithmes; en complexité on s'intéresse au temps (ou à l'espace de travail) nécessaire à leur résolution. Dans tous les domaines, on demande qu'il soit simple de déterminer si les entrées sont valides, de façon à ce que la difficulté éventuelle du problème soit contenue dans la question. En calculabilité, on demande en général que les entrées soient récursives (décidables), en complexité on ajoute à cela que les entrées soient faciles à déterminer (par exemple qu'elles soient plynômiales).

Définition 1.3.1. Un problème de décision est *résolu* (par programmes) s'il existe un programme qui décide si l'entrée est valide ou non, et s'il existe un programme qui, pour les entrées valides, retourne 1 si la réonse est OUI et retourne 0 si la réponse est NON. Dans ce cas on dit que le problème est *décidable*. Si il n'existe pas de tels programmes, on dira que le problème est *indécidable*.

De façon équivalente, un problème (de décision) est décidable si et seulement si l'ensemble de ses entrées valides et l'ensemble de ses instances positives sont des ensembles récursifs. Les problèmes PREMIER et CONNEXE sont clairement décidables.

On définit maintenant un exemple important de problème de décision qu'on appelle *Problème de l'arrêt* qui quant à lui est un exemple de problème indécidable. Ce problème prend en entrée le code d'un programme et d'une entrée pour ce programme. Le problème demande si le programme converge sur cette entrée.

ARRET :
Entrée: $\langle x, y \rangle$
Question: $[x|y] \downarrow$?

Évidemment, on ne peut pas juste lancer le programme x sur l'entrée y et attendre de voir ce qui se passe : si ce programme converge on ne peut a priori pas savoir combien de temps cela va prendre. Il se trouve qu'on ne peut rien faire d'autre, et cette impossibilité correspond à l'énoncé du théorème suivant.

Théorème 1.3.2 (Théorème de l'arrêt)**.** *Il n'existe pas de programme qui résoud le problème de l'arrêt. Autrement dit,* ARRET *est indécidable.*

On donne plusieurs formulations équivalentes de ce résultat :

- soit $f : \mathbb{N} \to \mathbb{N}$ la fonction telle que $f(\langle x, y \rangle) = 1$ si $[x|y] \downarrow$, et $f(\langle x, y \rangle) = 0$ sinon. Cette fonction n'est pas calculable. Attention ici : la définition de f est bien une description de la fonction et non un programme. Les termes *si* et*sinon* utilisés ici sont des conditions logiques et non pas des instructions de programmes.
- L'ensemble $A = \{\langle x, y \rangle, [x|y] \downarrow\}$ n'est pas récursif.

Démonstration. On suppose que f est calculable avec avec comme objectif de trouver une contradiction quelque part. Par la définition de fonction calculable, il existe donc un programme a tel que pour tout entrée $\langle x, y \rangle$ renvoie en sortie 1 si $[x|y] \downarrow$ et 0 sinon.

On construit maintenant le programme b suivant qui a une seule entrée x.

$b\langle x \rangle$:

```
if [a|⟨x, x⟩] = 0 then :
    return 23
else ⊥.
```

Que se passe-t'il quand on lance b sur l'entrée b?

Si le calcule converge, $[b|b] \downarrow$, alors le programme a exécuté une instruction `return` , donc nécessairement $[a|\langle b, b \rangle] = 0$. Mais par définition de a, quand il rend 0 c'est que $[b|b] \uparrow$. C'est absurde, donc nécessairement b doit diverger sur l'entrée b.

Examinons donc le cas où $[b|b] \uparrow$. D'après la définition de b, nous sommes donc dans le `else` . On en déduit que $[a|\langle b, b \rangle] \neq 0$ et du fait que a est totale et répond soit 0 soit 1, on en déduit que $[a|\langle b, b \rangle] = 1$. Mais dans ce cas par définition de a on a que $[b|b] \downarrow$, ce qui est absurde.

Donc notre hypothèse de départ mêne à une contradiction. Elle est donc fausse et on obtient que f n'est pas calculable. □

Donnons un autre exemple d'ensemble non récursif.

Définition 1.3.2. On appelle *ensemble de Kleene* l'ensemble $\mathbb{K} = \{x,\ [x|x]\downarrow\}$.

En réutilisant exactement la même preuve, on obtient que l'ensemble de Kleene n'est pas calculable. En effet dans la preuve nous n'avons appelé le programme a que sur sa diagonale $x = y$ (on a testé $[a|\langle b, b\rangle]$). Ainsi, si $\mathbb{K}$était récursif alors on pourrait faire la même preuve.

En revanche, $\mathbb{K}$ est énumérable. En effet, soit a le programme suivant.

$a\langle x\rangle$:

```
if [u|⟨x, x⟩]↓ then
    return 0
```

où u est le programme universel (Théorème 1.3.1). On aurait d'ailleurs pu écrire `if` $[x|x]\downarrow$ `then` ... On remarque que $\mathbb{K} = dom[a|\cdot]$, c'est-à-dire que c'est le domaine de convergence de a. L'ensemble de Kleene est donc énumérable.

Proposition 1.3.3. $\overline{\mathbb{K}}$ *n'est pas énumérable.*

Proof. D'après le Théorème de Post, si un ensemble et son complémentaires sont énumérables alors l'ensemble est récursif. L'ensemble $\mathbb{K}$ est énumérable donc si $\overline{\mathbb{K}}$ était énumérable alors ce théorème nous dirait que $\mathbb{K}$ serait récursif, ce qui est impossible d'après le théorème de l'arrêt. □

On va maintenant donner un autre exemple de problème indécidable de façon à généraliser notre résultat d'indécidabilité à d'autres problèmes. Cette généralisation se fait par une méthode qui s'appelle la réduction entre prooblèmes, méthode qui va émerger petit à petit.

ARRET$_0$:
Entrée: x
Question: $[x|0]\downarrow$?

Proposition 1.3.4. ARRET$_0$ *est un problème indécidable.*

Preuve. Soit c le programme suivant.

$c : \langle x, y\rangle \longrightarrow$ `if` $[x|x]\downarrow$ `then return` 0.

Remarquons que ici, la deuxième entrée y n'est pas utilisé à l'intérieur du programme, donc la sortie est inchangée si on la fait varier. Imaginons que l'on "fixe x comme un paramètre du programme". C'est-à-dire que pour chaque x, on considère le (numéro du) programme c_x qui prend en entrée y et renvoie la même chose que $[c|\langle x, y\rangle]$. Il y a deux types de tels programmes c_x, ceux qui retournent 0 sur chaque entrée y (lorsque $[x|x]\downarrow$), et ceux qui plantent sur toute entrée y (lorsque $[x|x]\uparrow$). Donc ARRET$_0(c_x)$ = True si et seulement si $x \in \mathbb{K}$.

On pourrait se convaincre qu'il existe un programme qui calcule la fonction $z \mapsto c_z$. En effet, il suffit de prendre le texte de c, et écrire le programme qui prend en entrée uniquement y, et où on remplace dans le texte chaque apparition de x par la valeur de l'entrée z.

Si un programme b pouvait décider ARRET_0, on pourrait s'en servir pour décider $\mathbb{K}$, ce qui n'est pas possible puisque cet ensemble n'est pas récursif. En effet, on prendrait x en entrée, on calculerait c_x, et on calculerait $[b|c_x]$. Or nous avons montré plus haut une équivalence qui nous dit que ce résultat de $[b|c_x]$ serait exactement la fonction caractéristique de $\mathbb{K}$. □

Pour bien comprendre cette preuve il faut en voir l'architecture globale : nous avons montré que si jamais on pouvait résoudre le problème ARRET_0, alors on pourrait se servir de ce programme de résolution de ARRET_0 pour résoudre $\mathbb{K}$. Ce qui n'est pas possible, ce dernier étant indécidable par le théorème de l'arrêt.

Cependant, une petite partie de notre preuve semble un peu problématique, ou au moins un peu différente. C'est notre affirmation qu'on peut calculer $z \mapsto c_z$. En effet, notre raisonnement n'est pas général mais propre à un modèle de calcul. Nous allons éclaircir la situation avec le théorème suivant, qui est un théorème propre à chaque modèle de calcul de même que le théorème d'universalité l'était.

Théorème 1.3.5 (Théorème de réification, Théorème SNM)**.** *Pour tout entiers positifs n, m, il existe une fonction calculable totale $S_n^m : \mathbb{N}^{m+1} \to \mathbb{N}$ tel que*

$$[a|\langle x_1, \ldots, x_m, y_1, \ldots, y_n\rangle] = [S_n^m\langle a, x_1, \ldots, x_m\rangle \mid \langle y_1, \ldots y_n\rangle].$$

Dans chaque système de programmation l'idée de la preuve est de fixé les valeurs de $x_1, \ldots, x_m$ dans le code de a et écrire l'algorithme qui produit ce code.

Preuve à adapter à chaque système de programmation. Pour calculer $S_n^m\langle a, x_1, \ldots, x_m\rangle$ on procède de la façon suivante. On commence par calculer la chaîne de caractère T du programme (de numéro) a. On fait l'hypothèse que dans T aucunes des variables $z_1, \ldots, z_m$ ne sont utilisées (sinon on en choisit d'autres), et on modifie T en remplaçant chaque occurrence de x_i par z_i, puis en rajoutant au tout début la ligne suivante:

$$z_1 = x_1; z_2 = x_2; \ldots; z_m = x_m;$$

On calcule ensuite le numéro du programme ainsi obtenu et on renvoie cette valeur. □

Par exemple si p est le numéro d'un programme qui calcule la fonction $(x, y) \mapsto x^2 + (x+1)y$ alors $S_1^1\langle p, 3\rangle$ est le numéro d'un programme qui calcule la fonction $y \mapsto 9 + 4y$.

Remarque culturelle. Les fonctions S_n^m sont issus de programmes différents dans chaque modèle de calcul mais qui sont très simples dans leur structure. Nul n'est besoin de recourir à des boucles ayant de subtiles conditions d'arrêt. On travaille juste sur un texte et on peut se passer de boucle `while` . De tels programmes simples sont appelés *récursifs primitifs*.

Application. Grâce à cette fonction on peut facilement exprimer la preuve d'indécidabilité de ARRET_0. Comme avant, on considère le programme à deux variables: Soit c le programme suivant $c\langle x, y\rangle$: `if` $[x|x] \downarrow$ `then return` 0. Pour tout x, le programme numéro $S_1^1\langle c, x\rangle$ réalise la fonction suivante $y \mapsto g(y)$ définie par : si $[x \mid x]$ converge alors $g(y) = 0$ et sinon $g(y) = \perp$. En particulier $[S_1^1\langle c, x\rangle)|0]$ converge si et seulement si $[x \mid x]$ converge. Si il existait

un algorithme pour décider la convergence en 0, on pourrait décider l'appartenance à $\mathbb{K}$, ce qui n'est pas possible puisque $\mathbb{K}$ n'est pas récursif.

Dans la prochaine section, nous allons systématiser et formaliser ce raisonnement en introduisant la notion de *réduction*.

1.3.4 Système acceptable de programmation

Dans ce cours, nous considérons un système de programmation, ayant les caractéristiques suivantes:

- il est capable d'énumérer les programmes grâce à leur numéro;
- il peut simuler un programme a sur une entrée x. On écrit $[a|x]$
- (universalité) $\exists u[u \mid \langle a, b\rangle] = [a|b]$
- il existe une fonction `step` permettra de simuler l'exécution de n'importe quel programme pendant un temps fixé (universalité décomposée en unités de temps)
- théorème SMN: $[a \mid \langle x, y\rangle] = [S_1^1\langle a, x\rangle \mid y]$
- opérations de base (tests, boucles, variables, etc.)

Par exemple : C++, python, et la majorité des systèmes de programmation que vous utilisez. Un tel système n'utilise pas "d'oracle". La notion d'oracle est très intéressante et centrale en calculabilité (mais aussi utilisée en algorithmique) mais ne sera pas étudiée dans cette partie du cours. Nous allons en donner quelques idées vagues à titre culturel. Un oracle est une information supplémentaire qu'on donne à un programme sous la forme d'une suite infinie de 0 et de 1 qu'il peut consulter à loisir. Bien sûr, si l'oracle est récursif, il n'apporte aucune puissance au système de programmation. Mais s'il ne l'est pas (un exemple intéressant d'oracle est $\mathbb{K}$) il apporte plus de puissance au calcul.

Théorème 1.3.6 (Axiomatisation de Shen $\approx$ 1990)**.** *Tout système acceptable de programmation correspond au calcul avec un oracle fixé, qui dépend du système considéré.*

Si en revanche on exclue les calculs avec oracles qui sont plus puissants que les calculs sans oracles, on obtient notre monde habituel qu'on appelle souvent le "calcul Turing". Dans ce cas, le théorème suivant nous montre que tous les systèmes acceptables de programmation sont exactement les mêmes, ce qui légitime notre choix de nous abstraire d'un système acceptable de programmation précis (comme les machines de Turing, les programmes Lisp, qu'on retrouve dans nombre de livres un peu datés).

Théorème 1.3.7 (Isomorphisme de Roger, $\approx$ 1960)**.** *Deux systèmes acceptables de programmation quelconques du calcul Turing sont isomorphes.*

Ce théorème présenté lui aussi à titre culturel a une signification importante qu'on peut retenir : on considère ici deux systèmes acceptables de programmation A et B de même puissance. On a énoncé le théorème pour le calcul Turing qui nous intéresse prioritairement, mais il est aussi valaple pour le calcul avec un oracle (le même pour les deux systèmes).

Le mot isomorphe signifie qu'il existe une correspondance bijective entre les deux espaces (iso-) et que la structure est préservée (-morphe). Ici la correspondance bijective est facile à comprendre : à un programme de A on associe un programme de B et réciproquement, de façon bijective. La préservation de la structure est ici plus subtile, on va faire cette bijection de telle façon qu'elle associe à un programme de A un programme de B qui calcule la même fonction (et réciproquement).

Par exemple, à un programme Python, on associe un programme en C et réciproquement, de façon bijective.

1.4 Réductions entre problèmes de décision

Les réductions que nous allons étudier dans ce cours s'appellent les réductions *many-one*. Il existe d'autres types de réduction comme les *réductions Turing* qui concernent le calcul avec oracle.

Définition 1.4.1. Soit A et B deux ensembles. On écrit $A \prec_m B$ si il existe une fonction f calculable totale, telle que

$$\forall x, x \in A \Leftrightarrow f(x) \in B \ .$$

On dit alors qu'il existe une *réduction many-one* de A vers B.

La fonction f n'est pas supposée être injective, d'où le nom "many-one": plusieurs éléments de A peuvent avoir la même image. La fonction n'est pas non plus supposée être surjective. C'est juste une fonction normale mais calculable et totale.

On interprète cette définition par *A n'est pas plus compliqué que B*. La raison de cette interprétation est la suivante : si on veut décider si un élément x est dans A ou pas, mais qu'on sait résoudre "$y \in B$?", alors on peut prendre cet élément x, le transformer par f et résoudre la question "$f(x) \in B$?"

Proposition 1.4.1. *$\prec_m$ est un préordre, c'est-à-dire:*

- symétrie $A \prec_m A$

- transitivité *si* $A \prec_m B$ *et* $B \prec_m C$ *alors* $A \prec_m C$

Anti-symétrie Attention, cette relation n'est pas anti-symétrique : on peut avoir des ensembles A et B différents qui sont en relation dans les deux sens. Par exemple, les entiers pairs et les entiers impairs. On dira que de tels ensembles sont *many-one équivalents* ce qu'on notera $A \sim_m B$. Une classe d'équivalence de cette relation s'appelle un *degré many-one*.

Pour simplifier les notations on écrira dans la suite du cours $\prec$ à la place de $\prec_m$.

Preuve. Pour montrer que $A \prec A$, il suffit de prendre la fonction identité $id : x \mapsto x$, qui est évidemment calculable totale. Pour la transitivité, soit f une fonction calculable totale telle que $\forall x, x \in A \Leftrightarrow f(x) \in B$ et g une fonction calculable totale telle que $\forall y, y \in B \Leftrightarrow g(y) \in C$. On définit la fonction $h = g \circ f$, c'est-à-dire la fonction h définie par $h(x) = g(f(x))$. Comme f est g sont récursives totales, h l'est aussi. Si $x \in A$, alors $y = f(x) \in B$ et comme $y \in B$ on a $g(y) \in C$. Donc $h(x) = g(f(x)) = g(y) \in C$. Inversement, si $x \notin A$, alors $f(x) \notin B$ et donc $g(f(x)) \notin C$, c'est-à-dire $h(x) \notin C$. On a donc montré que $A \prec C$. □

Exemple. On a $\mathbb{K} \prec \mathbb{K}_0$ où pour rappel $\mathbb{K} = \{x, [x|x] \downarrow\}$ et $\mathbb{K}_0 = \{x, [x|0] \downarrow\}$. Pour montrer cela, il faut donc trouver une fonction calculable f calculable totale telle que $\forall x, [x|x] \downarrow \Leftrightarrow [f(x)|0] \downarrow$. On commence par définir le programme suivant:

$$b\langle x, y\rangle : \texttt{if } [x|x] \downarrow \texttt{ then return } 4$$

et on pose f la fonction définie par $f(x) = S_1^1\langle b, x\rangle$. Cette fonction est bien calculable totale par le théorème SMN. Il faut maintenant montrer que $\forall x, x \in \mathbb{K} \Leftrightarrow S_1^1\langle b, x\rangle \in \mathbb{K}_0$. On commence par montrer $\Rightarrow$. Soit x tel que $x \in \mathbb{K}$, c'est-à-dire tel que $[x|x] \downarrow$. Le programme $f(x) = S_1^1\langle b, x\rangle$ renvoie donc 4 sur chaque entrée y. En particulier, il converge sur l'entrée $y = 0$, c'est-à-dire $[f(x)|0] \downarrow$, et donc $f(x) \notin \mathbb{K}_0$. Inversement, si x est tel que $[x|x] \uparrow$, alors le programme $f(x)$ diverge sur chaque entrée y. Donc en particulier il diverge sur l'entrée $y = 0$, c'est-à-dire $[f(x)|0] \uparrow$. Ainsi, $x \notin \mathbb{K}$ implique que $f(x) \notin \mathbb{K}_0$.

Nous allons maintenant montrer que cette relation de réduction se comporte bien avec les notions de récursivité

Proposition 1.4.2. *Soit A et B deux ensembles tels que $A \prec B$. Alors,*

- *si B est récursif alors A l'est aussi.*
- *si B est énumérable alors A l'est aussi.*

Preuve. La preuve du premier point a déjà été faite quand on a expliqué pourquoi cette relation signifie "pas plus compliqué que".

Pour le second point, imaginons que B soit énumérable. Alors il existe un programme b dont B est le domaine : $B = dom[b|\cdot]$. Construisons maintenant à l'aide de la fonction de réduction entre A et B notée f un programme a dont A est le domaine. On va prendre x en entrée et calculer $[b|f(x)]$. On sait que $f(x) \in B$ si et seulement si $x \in A$. On en déduit donc que $[a|x]$ s'arrête si et seulement si $x \in A$, ce que nous devions montrer. □

On utilise maintenant cette notion de réduction pour montrer le résultat suivant qui donne un outil facile pour démontrer l'indécidabilité d'un certain type d'ensemble.

Théorème 1.4.3 (Théorème de Rice)**.** *Soit $\mathcal{C}$ une propriété sur les fonctions partielles de $\mathbb{N}$ dans $\mathbb{N}$. Soit $P_{\mathcal{C}} = \{x, [x \mid \cdot] \in \mathcal{C}\}$. Alors, $P_{\mathcal{C}}$ est soit trivial (c'est-à-dire $P_{\mathcal{C}} = \emptyset$ ou $P_{\mathcal{C}} = \mathbb{N}$), soit $P_{\mathcal{C}}$ est indécidable.*

Attention : $\mathcal{C}$ est une propriété sur les *fonctions* et non sur les *programmes* !

Des exemples de telles propriétés sont: "être totale (définie partout)", "être définie sur les entrées paires", "avoir comme domaine de définition l'ensemble A, où A est un ensemble fixé énumérable".

Quelques cas plus subtils : "être calculable", "avoir un domaine de définition énumérable", "avoir comme domaine de définition l'ensemble B, où B est un ensemble fixé non énumérable". On montre facilement que dans ces derniers cas, $P_{\mathcal{C}}$ est trivial.

Démonstration du Théorème de Rice. On suppose que $P_{\mathcal{C}}$ est ni vide, ni égal à $\mathbb{N}$, et on va montrer que $P_{\mathcal{C}}$ n'est pas récursif (c'est-à-dire indécidable). On commence par supposer que la

fonction définie nulle part notée $\perp\!\!\!\perp$ n'est pas dans $\mathcal{C}$. On traitera l'autre cas en échangeant $\mathcal{C}$ et $\overline{\mathcal{C}}$ (il faudra juste remarquer que $P_{\overline{\mathcal{C}}} = \overline{P_{\mathcal{C}}}$).

Pour montrer que $P_{\mathcal{C}}$ est indécidable, on va montrer que $\mathbb{K} \prec P_{\mathcal{C}}$. D'après la proposition précédente, et comme $\mathbb{K}$ est indécidable (théorème de l'arrêt), cela impliquera que $P_{\mathcal{C}}$ l'est aussi.

Pour montrer que $\mathbb{K} \prec P_{\mathcal{C}}$, on va définir une fonction f calculable totale telle que

$$\forall x, x \in \mathbb{K} \iff f(x) \in P_{\mathcal{C}}$$

de la façon suivante. Notons a un programme quelconque de $P_{\mathcal{C}}$. C'est possible car $P_{\mathcal{C}}$ est non trivial donc non vide. Soit p le numéro du programme suivant

$$p : \langle x, y\rangle \mapsto \texttt{ if } [x|x]\downarrow \texttt{ then } \texttt{ return } [a|y]$$

On pose alors f la fonction définie par $f(x) = S_1^1\langle b, x\rangle$.

On montre maintenant que $x \in \mathbb{K} \iff f(x) \in P_{\mathcal{C}}$. Prenons $x \in \mathbb{K}$, dans ce cas, en examinant le programme p on s'aperçoit que $f(x)$ est un programme qui calcule la même fonction que a, donc qui est dans $P_{\mathcal{C}}$. Pour prouver la réciproque, utilisons la contraposée et montrons que si $x \notin \mathbb{K}$ alors $f(x) \notin P_{\mathcal{C}}$. En effet dans ce cas le programme $f(x)$ ne s'arrête pas et calcule donc la fonction $\perp\!\!\!\perp$. On a supposé au début de la preuve que cette fonction n'était pas dans $\mathcal{C}$ donc $f(x) \notin P_{\mathcal{C}}$.

□

Exemples

- Soit f_0 une fonction calculable alors $\{x, [x|\cdot] = f_0\}$ est indécidable. (Si f_0 est non calculable alors cet ensemble est vide)
- $\mathcal{C} = \{f,\ a \in dom f\}$. $P_{\mathcal{C}} = \{x,\ [x|a]\downarrow\}$. $P_{\mathcal{C}} \neq \emptyset$ et $P_{\mathcal{C}} \neq \mathbb{N}$, donc $P_{\mathcal{C}}$ est indécidable.
- $A_{a,b} = \{x,\ [x|a] = b\}$. $\mathcal{C} = \{f,\ f(a) = b\}$, $A_{a,b}$ est non trivial donc est indécidable.

Méthodologie Pour montrer avec le théorème de Rice qu'un ensemble A n'est pas récursif, il faut tout d'abord trouver une propriété $\mathcal{C}$ telle que $A = P_{\mathcal{C}}$. Ensuite on montre que A n'est pas trivial en exhibant un programme dans A et un autre à l'extérieur de A. Et hop, c'est bon.

Attention: on ne peut pas toujours utiliser ce théorème pour montrer l'indécidabilité d'ensembles. par exemple $A = \{x, [x|2x]\downarrow\}$ n'est pas récursif, mais on ne peut pas le démontrer en utilisant le théorème de Rice, car on ne peut pas trouver de propriété $\mathcal{C}$ telle que $A = P_{\mathcal{C}}$. En effet, $\mathcal{C}$ doit être une propriété sur les *fonctions* réalisées par les programmes et non les programmes eux-mêmes.

1.4.1 Propriétés des fonctions many-one

Dans cette section on donne donne propriétés des réductions many-one.

Proposition 1.4.4. *Soient $A \subseteq \mathbb{N}$ et $B \subseteq \mathbb{N}$ deux ensembles.*

1. $A \prec B \iff \overline{A} \prec \overline{B}$
2. *Si* $A \prec \emptyset$ *alors* $A = \emptyset$
3. *Si* $A \prec \mathbb{N}$ *alors* $A = \mathbb{N}$
4. *Si* B *est non trivial et* A *est décidable, alors* $A \prec B$
5. *Si* A *est énumérable, alors* $A \prec \mathbb{K}$

Preuve.

1. Il suffit de montrer un seul sens car $\overline{\overline{A}} = A$. Pour montrer que $A \prec B \Rightarrow \overline{A} \prec \overline{B}$, il suffit d'utiliser la même fonction f. En effet, supposons que $x \in \overline{A}$, c'est-à-dire que $x \notin A$. Comme $A \prec B$, on en déduit que $f(x) \notin B$, c'est-à-dire que $f(x) \in \overline{B}$. Inversement, si $x \notin \overline{A}$ alors $x \in A$ et donc $f(x) \in B$ ce qui se réécrit $f(x) \notin \overline{B}$.

2. Cela découle simplement du fait que la fonction f (dans la définition de réduction) doit être totale. Si A contenait un entier x alors on aurait $f(x) \in \emptyset$ ce qui est absurde.

3. Supposons que $A \prec \mathbb{N}$. Alors $\overline{A} \prec \overline{\mathbb{N}} = \emptyset$ d'après 1. Ainsi en appliquant 2., on déduit que $\overline{A} = \emptyset$, c'est-à-dire que $A = \mathbb{N}$. On pourrait aussi reprendre la même preuve que précédemment.

4. Comme B est non trivial, il existe donc deux éléments b et b' tels que $b \in B$ et $b' \notin B$. On pose f la fonction définie par $f(x) = b$ si $x \in A$ et $f(x) = b'$ sinon. Cette fonction est récursive totale (c'est facile de la programmer quand on connait b et b', en utilisant le test $x \in A$ qui est calculable).

5. Soit a un programme dont l'ensemble de convergence est A. On définit une fonction f par $f(x) = S_1^1\langle p, x\rangle$ où b est le numéro du programme suivant:

$$p : \langle x, y\rangle \mapsto \texttt{ if } [a|x]\downarrow \texttt{ then return } 23$$

La fonction f est calculable totale par le théorème SMN, et est telle que $\forall x, x \in A \Leftrightarrow f(x) \in \mathbb{K}$. En effet, soit $x \in A$, alors $f(x)$ est le numéro du programme qui converge partout vers 123, en particulier il converge sur l'entrée $f(x)$, c'est-à-dire $[f(x)|f(x)]\downarrow$. Inversement, si $x \notin A$, alors $[a|x]\uparrow$ et le programme $f(x)$ diverge sur chacune de ses entrées, donc en particulier sur l'entrée $f(x)$. On a donc montré que $A \prec \mathbb{K}$.

□

Montrer que A est énumérable

Methode 1 : par le domaine. On écrit un programme a qui converge exactement sur les x qui sont dans A.
Methode 2 : par l'image. On écrit un programme e tel que pour tout x dans A, il existe une entrée n sur laquelle e converge et dont la valeur retournée est x.

Montrer que A est récursif

On écrit un programme d tel que pour tout x, le programme d converge sur l'entrée x, et la valeur retournée est 1 si $x \in A$ et 0 sinon.

Montrer que A n'est pas énumérable

Méthode : Réduction. On trouve un ensemble B non-énumérable tel que $B \prec A$. Le plus souvent il suffit de prendre $B = \overline{\mathbb{K}}$. Pour cela, on construit une fonction totale récursive f telle que $\forall x, x \in B \Leftrightarrow f(x) \in A$. Pour construire f, il faut dans la plupart des cas construire un programme $p\langle x, y\rangle$, et poser $f(x) = S_1^1\langle p, x\rangle$ (cf Théorème SMN). Ce programme p a souvent recours à un test " `if` $[x|x]\downarrow$" ou quand ça ne semble pas adéquat, à la fonction `step`.

Montrer que A n'est pas récursif

Méthode 1: Rice. On trouve une propriété $\mathcal{C}$ sur les fonctions telle que $A = \{x, [x|\cdot] \in \mathcal{C}\}$, et on montre que A est non trivial. Attention, dans certains cas cette méthode ne s'applique pas (par exemple pour montrer que $\{x, [x|x] = x\}$ est indécidable).

Méthode 2: Réduction. On trouve un ensemble B non-récursif tel que $B \prec A$. Le plus souvent il suffit de prendre $B = \mathbb{K}$ ou $B = \overline{\mathbb{K}}$.

Méthode 3: Post. On montre que soit A soit $\overline{A}$ n'est pas énumérable (cf la méthode à gauche).

1.5 Inséparabilité

Cette notion va être très importante en logique, mais elle permet aussi de montrer des résultats intéressants à propos des programmes.

Définition 1.5.1. Soient A et B deux ensembles d'entiers disjoints. On dit que A et B sont (récursivement) *inséparables* si il n'existe pas d'ensemble d'entiers R récursif tel que $A \subseteq R$ et $B \subseteq \overline{R}$.

Exemples.

- $\mathbb{K}$ et $\overline{\mathbb{K}}$ sont inséparables. En effet, si un tel R existait alors nécessairement $R = \mathbb{K}$ ou $\overline{\mathbb{K}}$, mais c'est impossible puisque $\mathbb{K}$ et $\overline{\mathbb{K}}$ ne sont pas récursifs.
- Plus généralement: si A n'est pas récursif, alors A et $\overline{A}$ sont inséparables. Le raisonnement est le même. Mais bien sûr, la notion n'est pas très intéressante pour un ensemble et son complémentaire.
- Un autre exemple pas très intéressant : si A est récursif, et A et B disjoint alors a et B sont séparables. En effet, if suffit de prendre $R = A$.

Théorème 1.5.1. *Il existe des ensembles énumérables inséparables.*

Là c'est beaucoup plus intéressant car si les deux ensembles sont énumérables, ils ne sont complémentaires l'un de l'autre que dans le cas où l'ensemble est récursif (théorème de Post).

Preuve. On considère $A = \{x,\ [x|x] = 0\}$ et $B = \{x,\ [x|x] = 1\}$. Ces ensembles sont disjoints et énumérables (exercice très facile). En revanche, ces ensembles ne sont pas récursifs (exercice facile).

Supposons qu'il existe un ensemble récursif R qui sépare A et B, c'est-à-dire tel que $A \subseteq R$ et $B \subseteq \overline{R}$. Comme R est récursif, il existe un programme (de numéro r) qui calcule sa fonction carractérisitique donc qui converge toujours, et rend 1 ou 0 suivant que l'entrée est dans R ou pas. Demandons-nous maintenant quelle est la valeur de $[r|r]$?

Si c'est 1, cela signifie que $r \in R$ par définition de r et aussi que $r \in B$ par définition de B. Mais ceci n'est pas possible par définition de R qui doit séparer A et B. Bzzzt contradiction.

Si c'est 0, cela signifie que $r \notin R$ par définition de r et aussi que $r \in 1$ par définition de A. Mais ceci n'est pas possible par définition de R qui doit séparer A et B. Bzzzt contradiction.

Donc nous sommes dans une situation bloquée : $[r|r]$ ne peut valoir ni 0 ni 1. C'est absurde et nou sremontons donc à notre dernière hypothése pour affirmer que cette hypothèse est fausse (puisqu'elle implique l'absurde). Nous concluons qu'il n'existe pas d'ensemble récursif R qui sépare A et B. □

Nous allons appliquer ce théorème pour obtenir un résultat de programmation assez fondamental : si on a un programme qui plante sur certaines entrées, il n'est pas toujours possible de le corriger en un programme qui ne plante jamais – si on ne veut pas changer les valeurs qu'il donne quand il converge.

Théorème 1.5.2. *Il existe une fonction calculable (partielle) qui ne peut pas être étendue en une fonction calculable totale.*

Pour rappel, on dit que g est une fonction qui *étend* f si le domaine de f est strictement inclu dans celui de g et si pour tout $x \in domf$ on a $g(x) = f(x)$.

Preuve. On montre que c'est le cas de la fonction f suivante définie par

- $f(x) = 0$ si $[x|x] = 0$,
- $f(x) = 1$ si $[x|x] = 1$,
- $f(x) = \perp$ sinon.

Cette fonction est calculable car il suffit d'exécuter $[x|x]$ et si jamais le calcul converge on teste le résultat. Imaginons que l'on puisse étendre f en g récursive *totale*. On définit le programme suivant:

$p : x \mapsto$

```
if g(x) = 0 then return 1
else if g(x) = 1 then return 0
else return 0.
```

La fonction g est totale donc notre programme p aussi et il rend uniquement des 0 et des 1. Il calcule donc une fonction caractéristique d'ensemble récursif qu'on va appeler R. Oh surprise ! R sépare A et B (avec les notations du théorème précédent) ! Pas possible donc f ne peut pas être étendue en une fonction calculable totale. □

1.6 Preuves par diagonalisation

Dans notre preuve du théorème de l'arrêt comme dans la preuve de non séparation, on a effectué une manipulation curieuse : on a exécuté un programme sur lui-même. Ce procédé est en réalité lié à une construction très puissante et fondamentale, qui constitue le fondement des mathématiques du 20ème sciècle.

On note $\mathcal{P}(\mathbb{N})$ l'ensemble des parties de $\mathbb{N}$.

Théorème 1.6.1 (Théorème de Cantor Ĩ870)**.** *Il n'existe pas de surjection de* $\mathbb{N}$ *dans* $\mathcal{P}(\mathbb{N})$*.*

Remarque : on énonce souvent ce résultat en disant qu'il n'existe pas de surjection de $\mathbb{N}$ dans $\mathbb{R}$. Ces deux énoncés sont en réalité équivalent car $\mathbb{R}$ et $\mathcal{P}(\mathbb{N})$ sont en bijection.

On fait correspondre à chaque sous-ensemble de $\mathbb{N}$ un mot infini sur l'alphabet $\{O, 1\}$ de la façon suivante: le n-ième caractère de ce mot est un 1 si et seulement si n est dans l'ensemble. Cette correspondance est bijective: à chaque suite infinie est associée un sous-ensemble de $\mathbb{N}$. Par exemple l'ensemble $\{0, 3, 8, 12, 13, \dots\}$ est associé à la suite $10010000100011\dots$. Il est donc équivalent de démontrer qu'il n'existe pas de surjection de $\mathbb{N}$ dans l'ensemble des mots infinis sur $\{O, 1\}$.

En logique, on appelle réel une suite infinie sur $\{O, 1\}$, ce qui simplifie grandement les notations.

$$
\begin{array}{lcccccccccccc}
h(0) = & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & \cdot & \cdot \\
h(1) = & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & \cdot & \cdot \\
h(2) = & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & \cdot & \cdot \\
h(3) = & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & \cdot & \cdot \\
h(4) = & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & \cdot & \cdot \\
h(5) = & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & \cdot & \cdot \\
h(6) = & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & \cdot & \cdot \\
h(7) = & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & \cdot & \cdot \\
h(8) = & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & \cdot & \cdot \\
h(9) = & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & \cdot & \cdot \\
 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{array}
$$

Figure 1.1: illustration de la preuve du théorème de Cantor

Preuve du théorème de Cantor. On suppose que l'on peut numéroter l'ensemble des mots infinis sur $\{0, 1\}$. On considère le mot $s = 1001000000\dots$ obtenu en inversant les valeurs de la diagonale bleue de la figure. Ce mot n'est pas présent dans la liste ! En effet, il n'est pas égal au premier mot puisque le premier bit est différent, il n'est pas égal au deuxième puisque le deuxième bit est différent, etc. Ecrivons maintenant cela de façon un peu plus formelle pour nous entrainer :

Supposons qu'il existe une telle surjection qu'on appelle h. On construit le mot s définie par $s[n] = \overline{h(n)[n]}$. La barre au dessus de la valeur est utiliser pour échanger les 0 en 1 et vice-versa. Par définition de h, il existe un entier (niveau de ligne dnas le tableau) n_s telle que $h(n_s) = s$, et en particulier $\forall i, h(n_s)[i] = s[i]$. Mais par définition de s on a $s[n_s] = \overline{h(n_s)[n_s]}$. On obtient une contradiction sur la cellule $i = n_s$ où on devrait observer à la fois la valeur 0 et la valeur 1. On déduit de cette contradiction que cette surjection h n'existe pas. □

On énonce maintenant une version légèrement plus forte que le théorème précédent dont la démonstration est similaire.

Théorème 1.6.2 (Théorème général de Cantor)**.** *Soit A un ensemble. Il n'y a pas de surjection de A dans $\mathcal{P}(A)$.*

Preuve du théorème général de Cantor. Supposons qu'une telle surjection existe et notons-la f. Intuitivement, cette fonction donne aux parties de A des numéros qui viennent de A. Considérons maintenant l'ensemble suivant qu'on appelle en général l'ensemble diagonnal :

$$D = \{a \in A,\ a \notin f(a)\}\ .$$

Si f est surjective, cet ensemble D a bien un numéro dans A. Précisément, il existe $d \in A$ tel que $f(d) = D$. La question qu'on se pose maintenant est de savoir si $d \in D$.

Si $d \in D$ regardons la définition de D, nous obtenons $d \notin f(d)$ mais $f(d) = D$ donc $d \notin D$. Bzzzt contradiction.

Si $d \notin D$ regardons la définition de D, nous obtenons $d \in f(d)$ mais $f(d) = D$ donc $d \in D$. Bzzzt contradiction.

On déduit de tout cela que notre hypothèse est fausse et qu'une telle surjection n'existe pas. □

1.7 Théorèmes de points fixes

Deux familles importantes d'utilisation de ces théorèmes. La première en logique où les points fixes jouent un rôle très important. L'autre pour la compilation de programme, pour la sémantique des programmes récursifs au sens de l'algorithmique, c'est-à-dire des programmes qui, dans leur définition, s'appellent eux-mêmes.

Ces théorèmes ont été inventés dans les années 1950 par Stephen Kleene, nous présentons ici une version plus compréhensible de Hartley Rogers et qui date de 1967.

Théorème 1.7.1 (Point fixe)**.** *Soit f une fonction calculable totale. Alors il existe un programme n tel que la fonction calculée par le programme n et par le programme $f(n)$ sont les mêmes. C'est-à-dire,*

$$\exists n, [n|\cdot] = [f(n)|\cdot]$$

Remarques : tout d'abord, attention à la formulation : d'abord on considère une fonction f, et ensuite on obtient n. Donc n dépend de la fonction f considérée (et plus loin on verra comment il en dépend). De plus, on peut ici interpréter f comme un transformateur de programmes. Ce théorème dit qu'il existe un programme qui calcule la même chose que le programme transformé. Attention, le théorème ne dit pas que $n = f(n)$, c'est-à-dire qu'il ne dit pas que les codes de ces deux programmes sont les mêmes, il dit que ces deux programmes font la même chose.

Preuve. Affirmons d'abord qu'il existe une fonction calculable totale g telle que $\forall x, [g(x)|\cdot] = [[x|x]|\cdot]$. On l'obtient comme d'habitude avec le théorème SNM mais pour être bien complets, montrons cela. On écrit le programme p suivant

$p : \langle x, y\rangle \mapsto$

$a \leftarrow [x|x]$

`return` $[a|y]$

et on définit g par $g(x) = S_1^1\langle p, x\rangle$. On sait que g est calculable totale d'après le théorème SMN.

Maintenant, considérons m un programme qui calcule $f \circ g$. C'est possible car f et g sont calculables et la composée de deux fonctions calculables est aussi calculable.

On pose finalement $n = g(m)$ et on obtient bien ce qu'on voulait :

$$[n|\cdot] = [g(m)|\cdot] = [[m|m]|\cdot] = [f \circ g(m)|\cdot] = [f(n)|\cdot] \ .$$

La première égalité correspond à la définition de n, la seconde à celle de g, la suivante à le définition de m et la dernière à celle de n à nouveau. □

Avant de voir des applications à ce théorème, on donne quelques résultats complémentaires.

Théorème 1.7.2 (Théorème de la récursion étendu aux fonctions calculables)**.** *Soit f une fonction calculable. Alors, $\exists n, [n|\cdot] = [f(n)|\cdot]$.*

Preuve. Idem. Il faut relire la preuve précédente lentement en vérifiant que chaque étape est encore correcte même si la fonction est seulement partielle. La seule chose qu'il faut comprendre c'est que si $[a|b]$ diverge, alors $[[a|b]|\cdot] = \perp\!\!\!\perp$. □

Théorème 1.7.3. *Soit f une fonction calculable. Un de ses points fixes peut être obtenu effectivement : il existe un programme qui prend en entrée le code d'un programme pour la fonction f et qui renvoie un de ses points fixes. Autrement dit, on note le code de f par x et on obtient qu'il existe une fonction récursive totale η tel que $\forall x, [\eta(x)|\cdot] = [[x|\eta(x)]|\cdot]$.*

Preuve. Reprenons à nouveau la preuve du théorème du point fixe. La fonction g ne dépend pas de f mais quand on choisit pour m un programme qui calcule $f \circ g$, cet entier m dépend récursivement de x (code de f). Ceci s'obtient par une application élémentaire du théorème SNM qu'à ce stade on sait très bien faire. Mais notre point fixe n c'est $g(m)$ et g est récursive totale. Donc on obtient que n dépend récursivement du code de f et on pose $n = \eta(x)$ – la fonction η est bien une fonction récursive totale. □

On définit $\Pi_f = \{n, [n|\cdot] = [f(n)|\cdot]\}$ l'ensemble des points fixes de la fonction f.

Proposition 1.7.4. *Soit f une fonction calculable. Alors, Π_f est infini.*

Proof. Supposons que $\Pi_f = \{a_1, \ldots, a_k\}$ soit un ensemble fini et essayons de trouver une contradiction. Soit b un programme tel que pour tout i, $[b|\cdot] \neq [a_i|\cdot]$. Attention, on ne calcule pas ici le programme b, mais on sait seulement qu'un tel programme existe, et il existe car on sait faire un nombre infini de programme qui calculent des choses différentes, par exemple des fonctions constantes.

On écrit le programme c suivant : et on définit une fonction g récursive totale définie par $g(x) = S_1^1\langle c, x\rangle$. D'après le théorème du point fixe, il existe un programme n tel que $[n|\cdot] = [g(n)|\cdot]$. Regardons ce que calcule $g(n)$. Il calcule soit la même chose que b si $n \in \{a_1, \ldots, a_k\}$,

soit la même chose que $f(n)$. Le premier cas n'est pas possible car n ne peut pas calculer la même chose que b, nous l'avons construit exprès ainsi. Le second cas n'est pas possible non plus car on obtiendrait un nouveau point fixe pour f.

On peut exprimer la fin de cette preuve de façon un peu plus directe. en considérant la fonction g totale et récursive qui prend en entrée x et rend b si $x \in \Pi_f$ et $f(x)$ sinon. □

1.7.1 Applications

On propose plusieurs applications du théorème des points fixes : l'existence de programmes particuliers dont la sortie est liée à leur numéro ainsi qu'une autre preuve du théorème de Rice vu précédemment.

Programmes auto-reproducteur Pour un langage de programmation fixé (par exemple C, Python, etc.) on dit qu'un programme écrit dans ce langage est *auto-reproducteur* si il affiche son propre code. On peut montrer assez simplement grâce aux points fixes que de tels programmes existent.

Proposition 1.7.5. *Il existe un programme autoreproducteur, c'est-à-dire un programme qui écrit sont propre code, c'est-à-dire* $\exists n, \forall y [n|y] = n$.

Proof. On définit le programme $b\langle x, y\rangle$: `return` x et on définit la fonction f telle que $\forall x, f(x) = S_1^1\langle b, x\rangle$. La fonction $[f(n)|\cdot]$ est la fonction constante égale à n. Par le théorème SMN, f est calculable totale. D'après le théorème du point fixe, il existe donc n tel que $[n|\cdot] = [f(n)|\cdot] = n$. □

Exemple en C. Considérons le programme suivant écrit dans le langage C par Christoph Dürr

```
main(){char*b="main(){char*b=%c%s%c; printf(b,34,b,34);}";printf(b,34,b,34);}
```

Pour comprendre cet exemple, il faut avoir quelques idées sur le langage C: l'expression `printf(b,34,b,34)` signifie "imprimer la chaîne `b`, et comme cette chaîne est paramétrée par un caractère, une chaîne et un autre caractère représentés par `%c%s%c`, les paramètres suivants de `printf` (*i.e.* `34,b,34`) leur sont substitués dans `b`". Pour finir de comprendre l'exemple il faut savoir que le numéro du caractère `"` en ASCII est justement 34.

Proposition 1.7.6. *Il existe un programme qui diverge sur son propre code et seulement là, c'est-à-dire :* $\exists n,\ dom[n|\cdot] = \mathbb{N} \setminus \{n\}$.

Preuve. On écrit le programme suivant

$$b : \langle x, y\rangle \mapsto \texttt{if}\ \ x \neq y\ \texttt{then}\ \ \texttt{return}\ \ 43\ \texttt{else}\ \ \perp$$

et on considère la fonction f définie par $f(x) = S_1^1\langle b, x\rangle$. D'après le théorème du point fixe $\exists n, [n|\cdot] = [f(n)|\cdot]$. Mais on sait ce que fait le programme $f(n)$: il donne 43 tout le temps sauf en n. □

Preuve alternative du Théorème de Rice. Nous allons ici remontrer le théorème de Rice de façon facile et directe, en utilisant le théorème du point fixe, sans réduction. Nous mettons ainsi en évidence la puissance de ces techniques de point fixe.

Soit $P_{\mathcal{C}} = \{x, [x|\cdot] \in \mathcal{C}\}$ un ensemble de programmes associé à une certaine propriété $\mathcal{C}$ sur les fonctions. On suppose que $P_{\mathcal{C}}$ est non-trivial et donc on peut prendre deux programems a et b vérifiant $a \in P_{\mathcal{C}}$ et $b \notin P_{\mathcal{C}}$.

Si $P_{\mathcal{C}}$ était récursif, alors la fonction f définie par "$f(x) = b$ si $x \in P_{\mathcal{C}}$ et $f(x) = a$ sinon" serait calculable totale. Notez qu'on l'a inventée en échangeant en quelque sorte l'ensemble et son complémentaire. D'après le théorème du point fixe on obtient $\exists n, [n|\cdot] = [f(n)|\cdot]$. Regardons ce que vaut $f(n)$ pour ce n donné par le théorème. Il ne peut valoir que a ou b.

S'il vaut a alors par définition de f, $n \notin P_{\mathcal{C}}$ mais dans ce cas n calcule la même fonction que a donc il est dans $P_{\mathcal{C}}$. Bzzzt contradiction.

S'il vaut b alors par définition de f, $n \in P_{\mathcal{C}}$ mais dans ce cas n calcule la même fonction que b donc il n'est pas dans $P_{\mathcal{C}}$. Bzzzt contradiction.