Chapitre 5 : Décidabilité et classes de complexité

1 Formalisme et outils mathématiques

1.1 Relations et fonctions

Définition 1.1

Une relation $\mathcal{R}$ de $X \times Y$ est dite :
- ***déterministe** dès lors que $\forall (x, y, y') \in X \times Y \times Y, ((x, y) \in \mathcal{R}$ et $(x, y') \in \mathcal{R}) \Rightarrow y = y'$,*
- ***totale à gauche** dès lors que $\forall x \in X, \exists y \in Y, (x, y) \in \mathcal{R}$.*

Définition 1.2

*Étant donnés deux ensembles X et Y, une **fonction partielle** f de X dans Y est une relation déterministe $\mathcal{R} \subseteq X \times Y$. Le **domaine de définition** de f est alors la projection sur X de $\mathcal{R}$, soit $dom(f) = \{x \in X \mid \exists y \in Y, (x, y) \in \mathcal{R}\}$. Pour $x \in dom(f)$, on note $f(x)$ l'unique élément $y \in Y$ tel que $(x, y) \in \mathcal{R}$.*

Définition 1.3

*Étant donnés deux ensembles X et Y, une **fonction totale** f de X dans Y est une relation déterministe et totale à gauche de $X \times Y$.*

Autrement dit, la différence entre une fonction partielle et une fonction totale est qu'une fonction partielle peut n'être pas définie sur tout son ensemble de départ.
Étant donné un ensemble Y tel que $\diamondsuit \notin Y$, on peut compléter[♣] une fonction partielle $f \in X \rightharpoonup Y$ en une fonction totale définie comme suit.

$$\left(\begin{array}{lll} X & \to Y \cup \{\diamondsuit\} & \\ x & \mapsto \left\{ \begin{array}{ll} f(x) & \text{si } x \in \text{dom}(f) \\ \diamondsuit & \text{sinon} \end{array} \right. & \end{array} \right)$$

Notation 1.4

Soient X et Y deux ensembles. On note alors
- *$X \to Y$ l'ensemble des fonctions totales de X dans Y ;*
- *$X \rightharpoonup Y$ l'ensemble des fonctions partielles de X dans Y.*

Exercice de cours 1.5

- La relation $\{(n^2, n) \mid n \in \mathbb{Z}\}$ de $\mathbb{Z} \times \mathbb{Z}$ est-elle déterministe ? totale à gauche ? une fonction partielle ? une fonction totale ?
- Mêmes questions pour la relation $\{(n^2, n) \mid n \in \mathbb{Z}\}$ de $\mathbb{N} \times \mathbb{Z}$.
- Mêmes questions pour la relation $\{(n^2, n) \mid n \in \mathbb{N}\}$ de $\mathbb{N} \times \mathbb{N}$.
- Mêmes questions pour la relation $\{(n, n^2) \mid n \in \mathbb{N}\}$ de $\mathbb{N} \times \mathbb{N}$.

♣. à ne pas confondre avec prolonger la fonction, qui reviendrait à ajouter à la relation des couples (x, y) avec $x \notin X$

1.2 Notion de problème

Définition 1.6

*Étant donnés deux ensembles $\mathcal{E}$ et $\mathcal{S}$, on appelle **problème** une relation $\mathcal{R} \subseteq \mathcal{E} \times \mathcal{S}$ totale à gauche, i.e. qui vérifie $\forall e \in \mathcal{E}, \exists s \in \mathcal{S}, (e,s) \in \mathcal{R}$. Les éléments de $\mathcal{E}$ sont appelés les **entrées** (ou **instances**) du problème et ceux de $\mathcal{S}$ sont appelés ses **sorties**.*

Remarque 1.7

- Un problème est donc vu comme l'ensemble des liens entrées/sorties qu'il définit.
- Un problème n'est pas nécessairement une fonction, même partielle. En effet, il peut exister plusieurs sorties associées à une même entrée. En revanche, pour chaque entrée, il faut qu'il existe au moins une sortie associée.

Notation 1.8

Lorsque Q est un problème, on note $\mathcal{E}_Q$ (resp. $\mathcal{S}_Q$) l'ensemble des entrées (resp. celui des sorties) du problème Q.

Exemple 1.9
Considérons par exemple le problème :

$$\textsc{Prime} : \begin{cases} \textbf{Entrée} & : & \text{Un entier } n \in \mathbb{N} \\ \textbf{Sortie} & : & n \text{ est-il premier ?} \end{cases}$$

Ce problème peut être représenté par l'ensemble de couples : $\{(0,\mathsf{F}),(1,\mathsf{F}),(2,\mathsf{V}),(3,\mathsf{V}),(4,\mathsf{F}),\ldots\} \subseteq \mathbb{N}\times\mathbb{B}$.

Exemple 1.10
Considérons le problème :

$$\textsc{Find}_0 : \begin{cases} \textbf{Entrée} & : & n\in\mathbb{N}, T \text{ un tableau de } n \text{ entiers contenant au moins un } 0 \\ \textbf{Sortie} & : & \text{Un entier } i \in [\![0, n-1]\!] \text{ tel que } T[i] = 0 \end{cases}$$

Ce problème peut être représenté par l'ensemble de couples suivant.

$$\{((3,\texttt{[1, 0, 1]}),1),((3,\texttt{[0, 0, 1]}),1),((4,\texttt{[0, 0, 0, 1]}),0),\ldots\}$$

Exercice de cours 1.11

Les relations suivantes sont-elles des problèmes ? Si oui les reformuler sous la forme **Entrée : ...**, **Sortie : ...**.

- $\{(n,n^2) \mid n \in \mathbb{N}\}$;
- $\{(n^2,n) \mid n \in \mathbb{N}\}$;
- $\{(n^2,n) \mid n \in \mathbb{Z}\}$;
- $\{(n,\mathsf{V}) \mid n \in C\} \cup \{(n,\mathsf{F}) \mid n \in \mathbb{Z} \setminus C\}$, où $C = \{n^2 \mid n \in \mathbb{N}\}$;
- $\{(n^2,n) \mid n \in \mathbb{Z}\}$ comme relation sur $C \times \mathbb{Z}$.

On préfère souvent la représentation d'un problème par la donnée classique des **Entrée/Sortie**.

Représentation des données. Considérons le problème de l'égalité de deux langages réguliers.

$$\textsc{Égalité} : \begin{cases} \textbf{Entrée} & : & \text{Deux langages réguliers } L_1 \text{ et } L_2. \\ \textbf{Sortie} & : & \text{A-t-on } L_1 = L_2 \text{ ?} \end{cases}$$

Le chapitre automates et langages réguliers nous assure qu'il est possible d'écrire un algorithme permettant de répondre à ce problème. Toutefois il est clair que l'algorithme que nous allons devoir mettre en place dépend fortement de la représentation initiale des deux langages. Sont-ils

donnés sous la forme d'expressions régulières ? Sous la forme d'automates déterministes complets ? Sous la forme d'automates avec ε-transitions ? Aussi afin de lever l'ambiguïté de la description **Entrée/Sortie** d'un problème, on précise parfois la représentation choisie pour les entrées/sorties.

Exemple 1.12
Le test de primalité d'un entier peut se décliner en l'un ou l'autre des deux problèmes ci-dessous. Il sera beaucoup plus simple de résoudre PRIME$_f$ que PRIME.

$$\text{PRIME} : \begin{cases} \textbf{Entrée} & : & \text{Un entier } n \in \mathbb{N}, \text{ représenté par la suite de ses bits en base } 2 \\ \textbf{Sortie} & : & n \text{ est-il premier ?} \end{cases}$$

$$\text{PRIME}_f : \begin{cases} \textbf{Entrée} & : & \text{Un entier } n \in \mathbb{N}, \text{ représenté par sa décomposition en facteurs premiers} \\ \textbf{Sortie} & : & n \text{ est-il premier ?} \end{cases}$$

Dans la suite, lorsque ce n'est pas précisé les entiers manipulés par les problèmes sont représentés en binaire. Ce sont donc des mots sur l'alphabet $\{0,1\}$.
On distingue dans la suite deux types de problèmes particuliers : les problèmes de décisions, qui sont au cœur de ce chapitre, et les problèmes d'optimisation. Ces derniers, déjà rencontrés pour les schémas algorithmiques qu'ils appellent à mettre en place, ne sont pas directement l'objet de ce chapitre, mais on montre qu'ils sont naturellement associés à des problèmes de décision.

1.3 Problème de décision

Définition 1.13

*On appelle **problème de décision** un problème $\mathcal{R} \subseteq \mathcal{E} \times \mathbb{B}$ tel que $\forall e \in \mathcal{E}, \exists! s \in \mathbb{B}, (e,s) \in \mathcal{R}$. Un tel problème peut donc être vu comme une fonction totale de $\mathcal{E}$ dans $\mathbb{B}$.*
*On appelle alors **instances positives** (resp. **instances négatives**) les entrées pour lesquelles la sortie est V (resp. F).*

Notation 1.14

Lorsque Q est un problème de décision, on note Q^+ (respectivement Q^-) l'ensemble de ses instances positives (respectivement négatives). On dit de Q que c'est un problème trivial dès lors que $Q^+ = \emptyset$ ou $Q^- = \emptyset$.

Exemple 1.15

Pour $L \subseteq \Sigma^\star$, APPARTIENT$_L$: $\begin{cases} \textbf{Entrée :} \ w \in \Sigma^\star \\ \textbf{Sortie :} \ \text{A-t-on } w \in L \text{ ?} \end{cases}$ est un problème de décision, et APPARTIENT$_L^+ = L$, APPARTIENT$_L^- = L^c$.

Définition 1.16

*Soit Q un problème de décision. Le **co-problème** associé à Q, ou **problème complémentaire**, est le problème de décision sur $\mathcal{E}_Q$ défini par $Co - Q \overset{\text{déf}}{=} \{(e,\overline{s}) \mid (e,s) \in Q\}$.*

Remarque 1.17

Informellement, le co-problème pose la question "inverse" du problème initial. Cela revient à intervertir le rôle des réponses V et F. On note donc que le co-problème du co-problème est le problème initial.

Exemple 1.18
Le problème de savoir si un entier est composé, *i.e.* le problème COMPOSÉ est le co-problème de PRIME.
Pour $L \subseteq \Sigma^\star$, le co-problème de APPARTIENT$_L$ est APPARTIENT$_{L^c}$.

1.4 Problème d'optimisation

Définition 1.19

Soit $Q \subseteq \mathcal{E} \times \mathbb{R}$ un problème dont les sorties sont des réels.
*S'il existe, pour toute entrée $e \in \mathcal{E}$, un ensemble $sol(e)$, et une fonction $c_e \in sol(e) \to \mathbb{R}$ tels que $\min\{c_e(s) \mid s \in sol(e)\}$ est bien défini et est la seule valeur associée à e selon Q, alors on dit que Q est un **problème de minimisation**.*

*En remplaçant $\min$ par $\max$ on obtient la définition d'un **problème de maximisation**.*
*Dans les deux cas, on parle de **problème d'optimisation***

Cette définition, donnée sous forme d'existence, peut recouvrir tous les problèmes de décision. Cependant, on retiendra qu'un problème d'optimisation est un problème qui définit, pour chaque entrée e, un maximum ou un minimum à calculer, et ce en fixant l'**ensemble des solutions** $\mathsf{sol}(e)$ et la **fonction objectif** c_e. On appelle alors **valeur optimale** pour l'entrée e la valeur $\mathrm{opt}\{c_e(s) \mid s \in \mathsf{sol}(e)\}$ (pour $\mathrm{opt} = \min$ ou $\mathrm{opt} = \max$ selon les cas), et **solution optimale** pour l'entrée e toute solution $s \in \mathsf{sol}(e)$ telle que $c_e(s)$ est la valeur optimale.

Remarque 1.20

On appelle aussi problème d'optimisation le problème associant à chaque entrée e, non pas la valeur optimale, mais toutes les solutions optimales.

Définition 1.21

*Le **problème de décision associé** à un problème d'optimisation Q_O est obtenu en ajoutant aux entrées de Q une valeur de seuil et en demandant s'il est possible ou non, de trouver une solution dont la valeur est meilleure que le seuil.*

Problème de minimisation Q_O
$$\begin{cases} \textbf{Entrée} & : & e \in \mathcal{E}_Q \\ \textbf{Sortie} & : & \min\limits_{s \in sol(e)} c_e(s) \end{cases}$$

Problème de décision associé Q
$$\begin{cases} \textbf{Entrée} & : & e \in \mathcal{E}_Q,\ K \in \mathbb{N} \\ \textbf{Sortie} & : & \text{Existe-t-il } s \in sol(e) \text{ tel que } c_e(s) \leqslant K \text{ ?} \end{cases}$$

Problème de maximisation Q_O
$$\begin{cases} \textbf{Entrée} & : & e \in \mathcal{E}_Q \\ \textbf{Sortie} & : & \max\limits_{s \in sol(e)} c_e(s) \end{cases}$$

Problème de décision associé Q
$$\begin{cases} \textbf{Entrée} & : & e \in \mathcal{E}_Q,\ K \in \mathbb{N} \\ \textbf{Sortie} & : & \text{Existe-t-il } s \in sol(e) \text{ tel que } c_e(s) \geqslant K \text{ ?} \end{cases}$$

Exemple 1.22
Dans le cas du problème du plus court chemin dans un graphe connexe, le problème de décision associé au problème d'optimisation présenté plus haut est le problème suivant.

$$\begin{cases} \textbf{Entrée} & : & \text{Un graphe orienté pondéré } G = (S, A, c), \text{ deux sommets } s \text{ et } t \text{ de } S, \text{ un seuil } K \in \mathbb{N} \\ \textbf{Sortie} & : & \text{Existe-t-il un chemin de } s \text{ à } t \text{ dans } G \text{ de longueur} \leqslant K \text{ ?} \end{cases}$$

Exercice de cours 1.23

On considère le problème du rendu de monnaie suivant : étant donnés un entier $n \in \mathbb{N}^\star$, une famille de pièces $(p_1, p_2, \ldots, p_n) \in \mathbb{N}^n$ tels que $p_1 = 1$, et un montant M à rendre, on cherche à trouver un rendu de monnaie pour le montant M, utilisant le moins de pièces possible. Par exemple pour le système monétaire $(1, 2, 5)$ il est possible de rendre le montant $M = 6$ au moyen de $2 + 2 + 2$ ou $5 + 1$, cette deuxième solution étant préférée. Formaliser le problème du rendu de monnaie comme un problème d'optimisation. Donner le problème de décision associé. Justifier que ces problèmes sont bien définis.

2 Décidabilité

Un problème n'est pas une unique question qui attend une réponse, un problème est une famille de questions. Ainsi la solution d'un problème est un outil permettant de calculer l'association entrée/-sortie, autrement dit un algorithme.

Définition 2.1

*Un algorithme $\mathcal{A}$ est dit **solution** d'un problème R dès lors que pour toute entrée $e \in \mathcal{E}_R$ l'exécution de l'algorithme $\mathcal{A}$ sur l'entrée e produit une sortie $s \in \mathcal{S}$ telle que $(e, s) \in R$, autrement dit telle que s est une sortie attendue pour l'entrée e.*

Afin de pouvoir distinguer les problèmes admettant des solutions algorithmiques de ceux qui n'en admettent pas, il va nous falloir définir plus formellement la notion d'algorithme. C'est l'objet de la section 2.1.

Remarque 2.2

Cependant, la définition d'algorithme comme "suite finie et non ambiguë d'une suite d'instructions et d'opérations élémentaires", bien qu'informelle, nous permet déjà des remarques de cardinalité. En effet, si les opérations élémentaires sont en nombre au plus dénombrable, alors il existera une quantité au plus dénombrable d'algorithmes. Or on peut exhiber une infinité non dénombrable de problèmes tels que deux de ces problèmes ne peuvent être résolus par le même algorithme, ce qui assure qu'il est impossible de répondre à tous les problèmes.

Démonstration : Afin de justifier ces affirmations, on considère la famille de problèmes ci-dessous.

Pour $x \in [0,1[$, $\textsc{Bit}_x$: $\begin{cases} \textbf{Entrée :} \ n \in \mathbb{N}^\star \\ \textbf{Sortie :} \ \text{le } n\text{-ième bit de la décomposition dyadique}^{a} \text{de } x \ ? \end{cases}$

Soit $x \in [0,1[$ et $y \in [0,1[$ avec $x \neq y$. Si $\mathcal{A}$ est une solution à la fois au problème $\textsc{Bit}_x$ et au problème $\textsc{Bit}_y$, alors pour tout n dans $\mathbb{N}$ le n-ième bit de l'écriture dyadique de x et y est le même : $\mathcal{A}(n)$, donc x et y ont la même écriture dyadique, ils sont donc égaux. Absurde. Sachant que $[0,1[$ est infini non dénombrable, on en déduit qu'il faudrait une infinité non dénombrable d'algorithmes pour résoudre tous les problème de la famille $(\textsc{Bit}_x)_{x\in[0,1[}$. Ainsi, il existe dans cette famille des problèmes qui n'admettent pas de solution algorithmique ♠. □

2.1 Modèle de calcul

Pour définir un modèle de calcul, il faut fixer :
- un ensemble de machines ;
- le comportement de ces machines sur une entrée, *i.e.* définir comment elles s'exécutent ;
- dire si une machine répond à l'issue de son exécution, et le cas échéant quelle est sa réponse.

Dans ce chapitre, conformément au programme, on se borne à l'étude de modèles de calcul **déterministes**, ce qui signifie que l'exécution d'une machine sur une entrée, et la réponse qui en découle, sont toujours les mêmes.

Exemple 2.3

Par exemple les automates déterministes complets forment un modèle de calcul déterministe :
- chaque machine est caractérisée par un automate $\mathcal{A}$;
- étant donné un mot w en entrée, la machine exécute $\mathcal{A}$ sur w, et cela mène à un état q ;

♡. On dit que $(a_i)_{i\in\mathbb{N}} \in \{0,1\}^{\mathbb{N}}$ est **l'écriture dyadique** de x dès lors que $\sum_{i\in\mathbb{N}} \frac{a_i}{2^{i+1}} = x$.

♠. Il en existe même une infinité non dénombrable.

- la machine retourne alors V ou F selon que q est final ou non dans $\mathcal{A}$.

Exemple 2.4
Les automates déterministes (même non complets) forment aussi un modèle de calcul déterministe :
- chaque machine est caractérisée par un automate $\mathcal{A}$;
- étant donné un mot w en entrée, la machine exécute $\mathcal{A}$ sur w, cela bloque (par manque de transition) ou mène à un état q ;
- la machine retourne F si l'exécution de $\mathcal{A}$ sur w bloque ou aboutit à un état non final dans $\mathcal{A}$ et V sinon (*i.e.* la lecture de w a mené sans blocage à un état final).

Exemple 2.5
Les automates (même non déterministes ou non complets) forment un modèle de calcul déterministe :
- chaque machine est caractérisée par un automate $\mathcal{A}$;
- étant donné un mot w en entrée, la machine envisage toutes les exécutions possibles de $\mathcal{A}$ sur w ;
- la machine retourne V dès lors que l'une de ces exécutions de $\mathcal{A}$ sur w aboutit à un état final, et F sinon.

Pour se convaincre que ce modèle de calcul est bien un modèle de calcul déterministe, on peut penser à la fonction codée lors du TP sur les automates afin de tester si un mot est reconnu par un automate quelconque. Si on fixe l'automate, deux appels de cette fonction sur le même mot se passent exactement de la même manière, de manière déterministe.

Remarque 2.6

> Puisqu'on a montré qu'un automate quelconque est équivalent à un automate complet déterministe, ce modèle de calcul est en fait équivalent aux deux précédents.

Il est clair que les automates répondraient de manière assez pauvre aux problèmes de décision auxquels on peut s'intéresser. Le lemme de l'étoile assure par exemple qu'ils ne permettent pas de tester si un mot est bien parenthésé …. En fait les automates ne représentent pas fidèlement l'"expressivité" de nos machines de calcul usuelles, à savoir les ordinateurs (qui nous permettent, entre autres choses, de tester si une chaîne de caractères est bien parenthésée à l'aide d'une pile).

Pour le reste du chapitre on fixe donc le modèle de calcul suivant.
- Une machine est donnée par une fonction OCaml de type `string -> string`.
- On exécute une telle machine sur une entrée qui est une chaîne de caractères comme s'exécute l'appel de fonction en OCaml, en supposant qu'on dispose d'une mémoire infinie, dans la pile et dans le tas.
- Un tel appel de fonction peut conduire à un comportement d'erreur (*i.e.* lève une exception ou entre dans boucle infinie), ou bien à une chaîne de caractères qui sera alors la réponse de la machine.

Exemple 2.7
La fonction `est_premier` ci-dessous est une machine de notre modèle de calcul.

```
let est_premier (s: string): string =
  let n = int_of_string s in
  if n <= 1 then "false"
  else
    let i       = ref 2 in
    let i_sq    = ref 4 in      (* inv : i_sq = i * i *)
    let compose = ref false in
    while not !compose && !i_sq <= n do
      if n mod !i = 0 then compose := true else ();
      i_sq := !i_sq + 2 * !i + 1;
      i    := !i + 1;
    done;
    string_of_bool (not !compose)
```

Cette machine a un comportement d'erreur sur les entrées qui ne sont pas des entiers de $\mathbb{Z}$.

Exercice de cours 2.8

> De quel problème la machine de l'exemple 2.1 est-elle solution ?

Remarque 2.9

> Bien que ce modèle de calcul soit un peu plus formel que la notion d'algorithme, il n'est pas aussi rigoureusement défini que l'est par exemple le modèle de calcul par automates. Un modèle plus formel, et couramment utilisé en décidabilité, est celui des machines de Turing. Ces machines sont une amélioration des automates au sens où, en plus de pouvoir lire, elles peuvent écrire. On n'en dira pas plus sur ce modèle car il est hors-programme, et surtout parce qu'il offre la même puissance de calcul que celui que l'on s'est donné ici.

Notation 2.10

> *Dans les résultats mathématiques (comprendre : en dehors des programmes OCaml) de la suite de ce chapitre, on désignera en fait l'ensemble des objets de type* `string` *par* $\Sigma^\star$ *et on désignera* `"true"` *et* `"false"` *respectivement par V et F.*

Notation 2.11

> *Dans la suite, lorsque* $\mathcal{M}$ *est une machine,* $w \in \Sigma^\star$ *et* $w' \in \Sigma^\star$*, on notera :*
> - $w \xrightarrow[\mathcal{M}]{} w'$ *pour désigner le fait que l'exécution de* $\mathcal{M}$ *sur l'entrée* w *conduit à la sortie* w' *;*
> - $w \xrightarrow[\mathcal{M}]{} \circlearrowleft$ *pour désigner le fait que l'exécution de* $\mathcal{M}$ *sur l'entrée* w *conduit à un comportement d'erreur (erreur non rattrapée, blocage ou boucle infinie).*

Où l'on rejette le problème du codage à plus tard. Les machines que nous venons de définir opèrent sur l'ensemble $\Sigma^\star$, elles ne sont donc pas directement en mesure de répondre aux problèmes opérant sur des ensembles plus "complexes", comme l'ensemble des graphes ou celui des langages reconnaissables par exemple. Toutefois il suffit d'exhiber une fonction de codage injective définissable en OCaml, de réciproque définissable en OCaml, $\varphi_{\mathcal{X}} : \mathcal{X} \to \Sigma^\star$ pour pouvoir travailler en OCaml sur un ensemble $\mathcal{X}$. On peut alors utiliser de telles fonctions de codage pour se ramener au contexte du cours.

```
let super_fonction (e: ℰ) : 𝒮 =
   ...

let super_fonction_avec_signature_convenable (e: string) : string =
  (φ_𝒮 (super_fonction (φ_ℰ^-1 e)))
```

Aussi on s'autorisera dans la suite à parler de machines opérant sur des ensembles d'entrées et de sorties autre que $\Sigma^\star$, pourvu que ceux-ci admettent des fonctions de codages. Nous reviendrons sur cette question dans la section 2.3.

Définition 2.12

> *Soit* $\mathcal{M}$ *une machine de signature* `string -> bool`*. On appelle* ***langage de*** $\mathcal{M}$*, l'ensemble des mots sur lesquels* $\mathcal{M}$ *s'arrête et retourne V, i.e.*
>
> $$\mathcal{L}(\mathcal{M}) = \{w \in \Sigma^\star \mid w \xrightarrow[\mathcal{M}]{} V\}.$$

Remarque 2.13

Le langage d'une machine $\mathcal{M}$ n'est pas nécessairement le complémentaire de $\{w \in \Sigma^\star \mid w \xrightarrow[m]{} \mathsf{F}\}$. En effet le comportement de $\mathcal{M}$ induit une partition de $\Sigma^\star$ qui contient aussi la partie $\{w \in \Sigma^\star \mid w \xrightarrow[m]{} \circlearrowleft\}$ représentant ses comportements d'erreur.

Exercice de cours 2.14

Donner le langage de chacune des machines `mystere1` et `mystere2` ci-dessous.

```
let mystere1 (s: string): bool =
  (String.length s = 0)
  || (
    let i = ref 0 in
    while s.[!i] <> 'a' do
      i := !i + 1
    done;
    true
  )
```

```
exception Found of int

let trouve_virgule (s: string): int option =
  (* calcule l'indice de la première virgule
     dans s, s'il en existe un, None sinon *)
  try
    for i = 0 to (String.length s - 1) do
      if s.[i] = ',' then raise (Found i)
    done;
    None
  with
  | Found i -> Some i

let mystere2 (s: string): bool =
  let Some(i) = trouve_virgule s in
  let x = int_of_string (String.sub s 0 i) in
  let y = int_of_string (String.sub s (i+1)
  ↪  (String.length s - i - 1)) in
  x = 2 * y
```

2.2 Décidabilité et calculabilité

2.2.1 Fonctions calculables

On dira d'une fonction partielle f qu'elle est "calculable" dès lors qu'il existe une machine dont l'exécution, sur toutes les valeurs e pour lesquelles f est définie, termine et vaut $f(e)$.

Définition 2.15

*Une fonction partielle $f \in \mathcal{E} \rightharpoonup \mathcal{S}$ est dite **calculée** par une machine $\mathcal{M}$ dès lors que :*

$$\forall e \in \mathit{dom}(f), e \xrightarrow[m]{} f(e).$$

*Une fonction partielle $f \in \mathcal{E} \rightharpoonup \mathcal{S}$ est dite **calculable** s'il existe une machine la calculant.*

Remarque 2.16

Cette définition ne spécifie pas le comportement d'une machine $\mathcal{M}$ calculant $f \in \mathcal{E} \rightharpoonup \mathcal{S}$ sur des entrées pour lesquelles f n'est pas définie.

Exemple 2.17

Considérons la fonction $\sqrt{\square} : \mathbb{Z} \rightharpoonup \mathbb{Z}$ définie sur $\mathrm{dom}(\sqrt{\square}) = \{p^2 \mid p \in \mathbb{N}\} \overset{\text{déf}}{=} C$ et telle que $\forall n \in C, \sqrt{n}$ est l'unique $r \in \mathbb{N}$ tel que $r^2 = n$. La machine `sqrt` ci-dessous calcule cette fonction partielle $\sqrt{\square}$.

```
let sqrt (n: int): int =
  if (n < 0) then
    -1
  else
    begin
      let i    = ref 0 in
      let i_sq = ref 0 in
      while (!i_sq <> n) do
        i_sq := !i_sq + 2 * !i + 1;
        i    := !i + 1
      done;
      !i
    end
```

On remarque que :
- sur le domaine de définition de $\sqrt{\square}$, `sqrt` et $\sqrt{\square}$ coïncident ;
- sur les entiers strictement négatifs (qui sont donc en dehors du domaine de définition de $\sqrt{\square}$) la machine retourne -1 ;
- sur les entiers positifs qui ne sont pas des carrés (qui sont donc en dehors du domaine de définition de $\sqrt{\square}$) la machine a un comportement de boucle infinie.

Exercice de cours 2.18

On considère la fonction partielle $f : \mathbb{Z} \rightharpoonup \mathbb{Z}$ définie sur $\text{dom}(f) = \{n \in \mathbb{N} \setminus \{0,1\} \mid n \text{ n'est pas premier}\}$ par $f(n) = \min\{p \in \mathbb{N} \mid p \text{ divise } n \text{ et } p \neq 1 \text{ et } p \neq n\}$. Démontrer que f est une fonction calculable.

2.2.2 Problèmes décidables

Puisqu'un problème de décision Q est un cas particulier de fonction totale de $\mathcal{E}_Q \rightharpoonup \mathbb{B}$, et donc un cas particulier de fonction partielle, on dit qu'il est décidé par une machine si celle-ci calcule cette fonction, ce qui conduit à la définition suivante.

Définition 2.19

Soit Q un problème de décision. Soit une machine $\mathcal{M}$.
*On dit que Q est **décidé par** $\mathcal{M}$ lorsque :*

$$\forall e \in \mathcal{E}_Q, (e \in Q^+ \Leftrightarrow e \xrightarrow[\mathcal{M}]{} V \text{ et } e \in Q^- \Leftrightarrow e \xrightarrow[\mathcal{M}]{} F)$$

*Un problème de décision est dit **décidable** s'il existe une machine le décidant.*

Remarque 2.20

Cette définition assure que la machine $\mathcal{M}$ décidant Q s'exécute sans comportement d'erreur sur toutes ses entrées. En effet, dans le cas d'un problème de décision, Q^+ et Q^- partitionnent $\mathcal{E}_Q$.

Exemple 2.21

Considérons le problème de décision suivant.

$$\textsc{Contient}_{\texttt{'a'}} : \begin{cases} \textbf{Entrée :} s \in \Sigma^\star \text{ une chaîne de caractères} \\ \textbf{Sortie :} \text{La chaîne } s \text{ contient-elle le caractère } \texttt{'a'} \text{ ?} \end{cases}$$

Ce problème est décidé par les deux machines ci-dessous.

```
let contient_a1 (s: string): string =
  let n = String.length s in
  let i = ref 0 in
  while (!i < n && s.[!i] <> 'a') do
    incr i;
  done;
  string_of_bool (!i < n)
```

```
let contient_a2 (s: string): string =
  let n = String.length s in
  let i = ref (n-1) in
  while (!i >= 0 && s.[!i] <> 'a') do
    decr i;
  done;
  string_of_bool (!i > (-1))
```

Exercice de cours 2.22

La machine `mystere1` de l'exercice de cours 2.14 décide-t-elle le problème $\textsc{Contient}_{\texttt{'a'}}$ défini dans l'exemple ci-dessus ?

Proposition 2.23

Soit Q un problème de décision. Q est décidable si et seulement si Co-Q l'est.

Démonstration : Si Q est décidable, il existe une machine $\mathcal{M}$ le décidant. On peut alors construire la machine $\mathcal{M}'$ qui s'exécute comme $\mathcal{M}$ puis renvoie la réponse inverse. En OCaml il suffit d'utiliser l'opérateur `not` sur la sortie. Cette machine $\mathcal{M}'$ décide Co-Q, donc Co-Q est décidable.

Puisque que Co-Co-Q = Q, le premier sens de l'implication suffit. □

Exercice de cours 2.24

Montrer que les problèmes suivants sont décidables.

$$\textsc{Pair} : \begin{cases} \textbf{Entrée} : s \text{ une chaîne de } \{0,1\}^\star \\ \textbf{Sortie} : s \text{ est-elle la représentation en base } 2 \text{ d'un entier pair ?} \end{cases}$$

$$\textsc{Impair} : \begin{cases} \textbf{Entrée} : s \text{ une chaîne de } \{0,1\}^\star \\ \textbf{Sortie} : s \text{ est-elle la représentation en base } 2 \text{ d'un entier impair ?} \end{cases}$$

2.2.3 Langages décidables

Pour définir la décidabilité d'**un langage** $L \subseteq \Sigma^\star$, on s'appuie sur le problème de décision qui consiste à tester l'appartenance d'un mot à L.

$$\textsc{appartient}_L : \begin{cases} \textbf{Entrée} & : & w \in \Sigma^\star \\ \textbf{Sortie} & : & \text{A-t-on } w \in L ? \end{cases}$$

Ainsi à tout langage est associé un problème de décision canonique. Réciproquement, à un problème de décision Q sur $\Sigma^\star$, on associe le langage de ses instances positives, *i.e.* $L = \text{Q}^+$. On a ainsi mis en bijection les langages sur Σ et les problèmes de décision sur $\Sigma^\star$.

Définition 2.25

Soit L un langage sur Σ. Soit $\mathcal{M}$ un machine.
*On dit que $\mathcal{M}$ **décide** L lorsqu'elle décide le problème $\textsc{appartient}_L$.*
*On dit ainsi que L est **décidable** s'il existe une machine décidant $\textsc{appartient}_L$.*

Remarque 2.26

De manière équivalente, on peut dire que le langage L est décidable lorsque sa fonction caractéristique est calculable. La fonction caractéristique $\mathbb{1}_L$ d'un langage L est la fonction totale de $\Sigma^\star$ dans $\mathbb{B}$ qui à un mot w associe V si $w \in L$ et F sinon.

Exercice de cours 2.27

Démontrer la remarque précédente.

Proposition 2.28

Un langage sur Σ est décidable si et seulement si c'est le langage d'une machine, opérant de $\Sigma^\star$ dans $\mathbb{B}$, qui termine sur toutes ses entrées.

Démonstration : Soit L un langage sur Σ. Si $\mathcal{M}$ est une machine opérant de $\Sigma^\star$ dans $\mathbb{B}$ et terminant sur toutes ses entrées, alors $\forall w \in \Sigma^\star, (w \xrightarrow[\mathcal{M}]{} \mathsf{V}$ ou $w \xrightarrow[\mathcal{M}]{} \mathsf{F})$. Ainsi on peut dire que L est décidable
si et seulement s'il existe une machine $\mathcal{M}$ de $\Sigma^\star$ dans $\mathbb{B}$ terminant sur toute entrée, telle que $\forall w \in \Sigma^\star, w \in L \Leftrightarrow w \xrightarrow[\mathcal{M}]{} \mathsf{V}$,
si et seulement s'il existe une machine $\mathcal{M}$ de $\Sigma^\star$ dans $\mathbb{B}$ terminant sur toute entrée, telle que $\forall w \in \Sigma^\star, w \in L \Leftrightarrow w \in \mathcal{L}(\mathcal{M})$
si et seulement s'il existe une machine $\mathcal{M}$ de $\Sigma^\star$ dans $\mathbb{B}$ terminant sur toute entrée, telle que $L = \mathcal{L}(\mathcal{M})$. □

Proposition 2.29

Un langage régulier est décidable.

Démonstration : En TD, nous avons construit une fonction `accepte : automaton -> string -> bool` permettant de tester, pour un automate $\mathcal{A}$ et un mot w si $w \in \mathcal{L}(\mathcal{A})$. Aussi étant donné un langage régulier L, par théorème de Kleene, il existe un automate $\mathcal{A}$ tel que $\mathcal{L}(\mathcal{A}) = L$, on considère alors la fonction `let accepte_l = (accepte` $\mathcal{A}$`)`, `accepte_l` est alors la fonction caractéristique de L. □

Remarque 2.30

Pour justifier qu'un langage (ou un problème) est décidable, il suffit d'assurer qu'il existe une machine le décidant, il n'est pas nécessaire d'expliciter cette machine, ni même de savoir comment la calculer.

Proposition 2.31

L'ensemble des langages décidables est stable par intersection, union, concaténation, complémentaire, et étoile de Kleene.

Démonstration : Soient L_a et L_b deux langages décidables, décidés par `decide_a : string -> bool` et `decide_b : string -> bool` respectivement. On peut alors affirmer que :

- $L_a \cup L_b$ est décidé par `fun s -> (decide_a s) || (decide_b s)`;
- $L_a \cap L_b$ est décidé par `fun s -> (decide_a s) && (decide_b s)`;
- $\overline{L_a}$ est décidé par `fun s -> not (decide_a s)`;
- $L_a \cdot L_b$ est décidé par la fonction qui suit.

```
let decide_c (s: string) : bool =
  let n = String.length s in
  let i = ref 0 in
  while !i <= n && not ( decide_a (String.sub s 0 !i) &&
                         decide_b (String.sub s !i (n- !i))) do
    incr i
  done;
  !i <> n+1
```

La preuve que $L_a^\star$ est aussi décidable est laissée en exercice. Pour la stabilité par complémentaire, on pouvait aussi remarquer que le problème $\textsc{Appartient}_{\overline{L}}$ est le problème $\textsc{Co-Appartient}_L$, lui-même décidable en tant que co-problème d'un problème décidable. □

Exercice de cours 2.32

Démontrer que $L_a^\star$ est aussi décidable.

Exercice de cours 2.33

Étant donné un mot $w = w_1 w_2 \dots w_p \in \Sigma^\star$, on lui associe son mot miroir $\overline{w} = w_p \dots w_2 w_1$.
Montrer que si L est un langage décidable alors son miroir $\overline{L}$ (défini comme $\{\overline{w} \mid w \in L\}$) est lui aussi décidable.
Montrer que si L n'est pas un langage décidable alors $\overline{L}$ n'est pas non plus décidable.

2.3 Sérialisation

Reprenons la discussion, laissée de côté ci-avant, sur la possibilité de coder des éléments de certains types au moyen de chaînes de caractères. Formalisons pour cela la notion de sérialisation.

Définition 2.34

*Soit `t` un type OCaml. On appelle **sérialisation calculable** du type `t`, la donnée d'une fonction OCaml `serialise : t -> string` telle que :*

- *`serialise` est injective ;*
- *pour tout élément `e` de type `t`, l'appel de `serialise` sur `e` termine sans erreur et `(serialise e)` est une chaîne bien parenthésée ;*
- *il existe une fonction `deserialise : string -> t option` telle que pour tout `s` de type `string`, l'appel `deserialise` sur `s` termine sans erreur et `(deserialise s)` vaut :*
  - *`None` si `s` n'est pas dans l'image de `serialise` ;*
  - *`Some(e)` si `s` est `(serialise e)` avec `e` de type `t`.*

*On dit qu'un type `t` est **sérialisable** s'il admet une sérialisation calculable.*

Remarque 2.35

On remarque si `t` est un type sérialisable, le problème de savoir si une chaîne de caractères `s` est la sérialisation d'un élément de type `t` est décidable. En effet il suffit d'appeler `(deserialise s)` et de tester si le résultat est différent de `None`.

Remarque 2.36

Lorsqu'un ensemble mathématique S admet une représentation "canonique" en OCaml au moyen d'un type `t`, on dira que S admet une sérialisation calculable dès lors que `t` en admet une.

Exemple 2.37

L'ensemble $\mathbb{Z}$, représenté en OCaml par le type `int`, admet une sérialisation calculable. En effet la fonction `string_of_int: int -> string` permet de sérialiser, et la fonction `intf_of_string_opt : string -> int option` de désérialiser. La fonction `int_of_string_opt` s'évalue à `None` lorsqu'elle est appliquée à une chaîne vide, ou contenant un point, ou des lettres ...♣. Elle renvoie aussi `None` lorsque la chaîne qu'on lui passe en entrée représente un entier trop grand pour la capacité du type `int`, mais dans notre modèle de calcul, à la mémoire non bornée, on suppose que le type `int` s'adapte pour pouvoir stocker des entiers arbitrairement grands.

Proposition 2.38

Si deux types `ta` et `tb` sont sérialisables, alors le type `ta * tb` admet aussi une sérialisation calculable.

Démonstration : Notons `serialise_a : ta -> string` et `serialise_b : tb -> string` les fonctions de sérialisations des types `ta` et `tb`. On considère alors la fonction de sérialisation suivante.

♣. soit exactement dans les cas où `int_of_string` lève l'exception `Failure "int_of_string"`

```
let serialise_couple ((a, b): ta * tb): string =
  "(" ^ (serialise_a a) ^ "),(" ^ (serialise_b b) ^ ")"
```

`serialise_couple` est injective. En effet, sachant que `(serialise_a a)` et `(serialise_b b)` sont bien parenthésées, on peut à partir de la chaîne `(serialise_couple (a, b))` identifier `(serialise_a a)` et `(serialise_b b)`. Ainsi, si `(serialise_couple (a, b)) = (serialise_couple (c, d))`, en identifiant, à la fois `(serialise_a a) = (serialise_a c)` et `(serialise_b b) = (serialise_b d)`. L'injectivité de `serialise_a` et `serialise_b` permet alors d'en déduire que a=c et b=d. La réciproque de cette fonction consiste justement à procéder à l'identification des sous-chaînes sérialisant la première et la deuxième composante du couple, puis à les déserialiser chacune. On peut la coder comme suit en OCaml.

```
let find_matching_closing_paren (i: int) (s: string) : int =
  (* hyp : 0 <= i < String.length s && s.[i]='(' *)
  (* calcule l'indice dans s de la parent. fermante associée à s.[i] *)
  let prof = ref 1       in
  let idx  = ref (i + 1) in
  let n = String.length s in
  while !prof <> 0 && (!idx < n) do
    if s.[!idx] = ')' then decr prof
    else if s.[!idx] = '(' then incr prof else ();
    incr idx
  done;
  if !prof <> 0 then failwith "parenthèse ouvrante non refermée";
  !idx-1
let deserialise_couple (s: string) : (ta * tb) option =
  let n = String.length s in
  try
    let len_a = (find_matching_closing_paren 0 s) - 1 in
    let len_b = (n-len_a-5) in
    Some((deserialise_a (String.sub s 1 len_a),
          deserialise_b (String.sub s (4+len_a) len_b)))
  with
  | Failure _ -> None
```

□

Exercice de cours 2.39

Pour la fonction de sérialisation `serialise_couple`, donner la sérialisation de $(1,(2,4))$.

Exercice de cours 2.40

Justifier que le type `string * int * int` est sérialisable en fournissant une fonction de sérialisation et de désérialisation. Pour la fonction proposée, quelle est la sérialisation de `("toto", 2, -3)`.

Proposition 2.41

Si un type t admet une sérialisation calculable, le type t `list` des listes d'éléments de type t admet une sérialisation calculable.

Démonstration : Soit `serialise_t : t -> string` une fonction de sérialisation du type `t`. Soit alors la fonction suivante.

```
let rec serialise_liste (s: t list): string =
  match s with
  | []     -> ""
  | p :: q -> "(" ^ (serialise_t p) ^ "),(" ^ (serialise_liste q) ^ ")"
```

Cette fonction est injective, pour les mêmes raisons que dans la démonstration précédente. La réciproque de cette fonction peut être définie en OCaml comme suit.

```
let rec deserialise_liste (s: string): t list option =
  if s = "" then Some []
  else
    match deserialise_couple s with
    | None -> None
    | Some((p, q)) ->
      match deserialise_t p, deserialise_liste q with
      | None, _ | _, None      -> None
      | Some(dt_p), Some(dl_q) -> Some(dt_p :: dl_q)
```

□

Exercice de cours 2.42

Pour la fonction de sérialisation `serialise_liste`, donner la sérialisation de `[2; 3; 4]` et de `[]`.
Donner un exemple de chaîne qui n'est pas désérialisable pour `deserialise_liste`.

Ces résultats sont plutôt encourageants. La plupart des objets manipulés en machine peuvent être décrits à l'aide de types produits ou de listes. On verra en TD comment sérialiser aussi les types sommes.

Sérialisation d'une machine. On peut remarquer qu'une machine elle-même peut être représentée par une chaîne de caractères bien parenthésée♣ : en effet c'est un programme OCaml, il suffit donc de le représenter par la chaîne des caractères qui composent son code source. Par exemple le programme ci dessous à gauche peut être représenté par la chaîne de caractères ci-dessous à droite.

```
let rec fact (n: int): int =
  if n = 0 then 1
  else n * (fact (n-1))
```

```
"let rec fact (n: int): int = if n = 0
↪  then 1 else n * (fact (n-1))"
```

Proposition 2.43

Les programmes OCaml admettent une sérialisation calculable au moyen de chaînes de caractères.

Notation 2.44

Lorsqu'on a affaire à une machine $\mathcal{M}$, on notera si besoin $\langle\mathcal{M}\rangle$ sa sérialisation.
Si de plus $\mathcal{M}$ termine sur l'entrée w, on notera $\mathcal{M}(w)$ le résultat.

2.4 Machine universelle

La possibilité de sérialiser une machine au moyen d'une chaîne de caractères qui peut elle-même être donnée en argument à une machine, nous pousse à nous interroger sur le pouvoir introspectif de notre modèle de calcul. L'idée est de savoir si, à partir de la sérialisation d'une machine $\mathcal{M}$ et d'un mot w, il est possible de calculer ce que $\mathcal{M}$ calcule sur l'entrée w. Autrement dit la question est de savoir si la fonction partielle♡ ci-dessous est calculable.

♣. Cette affirmation suppose que les parenthèses contenues entre des guillemets sont omises. En effet la syntaxe d'OCaml n'interdit pas la fonction `fun s -> s^")"`. Cependant on peut s'y ramener, quitte à ajouter des chaînes inutiles dans le code, par exemple en `fun s -> let _ ="(" in s^")"`.

♡. Fonction partielle à double titre : l'image d'un couple (s, w) n'est pas définie si s n'est pas la sérialisation d'une machine $\mathcal{M}$, et quand bien même, encore faut-il que l'exécution de cette machine $\mathcal{M}$ sur w termine.

$$\begin{cases} \Sigma^\star \times \Sigma^\star & \to \Sigma^\star \\ (\langle m \rangle, w) & \mapsto \text{Le résultat de l'exécution de } m \text{ sur } w \text{ si celle-ci termine sans erreur} \end{cases}$$

Remarque 2.45

L'écriture d'une machine universelle nécessite en premier lieu d'être capable de différencier les chaînes de caractères qui sont des sérialisations de programmes OCaml de celles qui n'en sont pas. Ce problème revient à l'écriture d'un analyseur lexical, syntaxique et d'un typeur pour les programmes OCaml. Aussi nous admettrons que le problème suivant est décidable.

$$\text{OCamlValide} : \begin{cases} \textbf{Entrée :} & w \in \Sigma^\star \\ \textbf{Sortie :} & w \text{ est-elle la sérialisation d'un programme OCaml valide ?} \end{cases}$$

Théorème 2.46

Il existe une machine, appelée ***machine universelle****, qui prend en entrée un couple de la forme* $(\langle m \rangle, w)$ *et qui calcule la sortie de la machine* m *sur* w *si celle-ci s'exécute sans erreur sur* w *et qui a un comportement d'erreur sinon.*

Démonstration : La preuve de ce théorème consiste en l'écriture d'un programme OCaml qui est capable de prendre en argument : une chaîne de caractères représentant un programme OCaml et une chaîne de caractères représentant son entrée et retournant le résultat de l'exécution du programme sur cette entrée. Un tel programme n'est rien d'autre que l'interprète OCaml que nous utilisons en TP. □

Notation 2.47

Dans les pseudo-codes, on notera m_{univ} *cette machine universelle. Dans les codes OCaml, on notera* `interprete` *la fonction de type* `string -> string -> string` *qui correspond.*

Remarque 2.48

Ce résultat (le théorème 2.46) n'est pas anecdotique et en dit long sur le pouvoir d'expressivité du modèle de calcul que l'on s'est donné. Le modèle de calcul que représentent les automates ne permet pas ce résultat.

Remarque 2.49

On note que m_{univ} ne termine pas nécessairement sur toutes ses entrées. Elle peut bloquer lorsque l'entrée n'est pas de la forme attendue pour la sérialisation d'un couple de la forme $(\langle m \rangle, w)$. Mais même si elle prend en entrée la sérialisation de $(\langle m \rangle, w)$, elle peut bloquer du fait que m peut elle-même bloquer sur w.

Exercice de cours 2.50

On fixe un entier $n \in \mathbb{N}^\star$, et deux mots $u \in \Sigma^\star$ et $v \in \Sigma^\star$.
Montrer que la fonction partielle suivante est calculable, et préciser son domaine de définition.

$$\begin{cases} \Sigma^\star & \to \mathbb{N} \\ \langle m \rangle & \mapsto |m(u)| + |m(v)| \end{cases}$$

Interprétation bornée. L'exécution d'un programme OCaml sur une entrée peut être découpée en la suite de ses étapes "élémentaires". Ce découpage est celui que nous avons introduit pour établir la notion de complexité algorithmique. Ayant admis qu'on peut coder en OCaml un interpréteur de code OCaml, on admet aisément qu'on peut aussi coder en OCaml un interpréteur qui incrémente

un compteur à chaque étape de calcul (à chaque opération élémentaire), et s'arrête au delà d'un certain nombre d'étapes. Autrement dit la fonction partielle suivante est calculable.

$$\begin{cases} \Sigma^\star \times \mathbb{N} \times \Sigma^\star \to \Sigma^\star \\ (\langle \mathcal{M} \rangle, n, w) \mapsto \text{Le résultat de l'exécution de } \mathcal{M} \text{ sur } w \text{ si celle-ci termine en moins de } n \text{ étapes} \end{cases}$$

Notation 2.51

Dans les codes OCaml, on notera `interprete_borne` *la fonction de type* `string -> int -> string -> string` *qui correspond.*

Un tel interpréteur s'arrête toujours, même si le code à interpréter contenait une boucle infinie, et peut rattraper les éventuelles erreurs pour répondre F. Ainsi le problème suivant est décidable.

ArrêtBorné :

Entrée : Une machine $\mathcal{M}$, un mot w et un entier n
Sortie : L'exécution de $\mathcal{M}$ sur w termine-t-elle sans erreur en moins de n étapes élémentaires ?

Exercice de cours 2.52

Démontrer que le problème suivant est décidable.

Entrée : Un mot $w \in \Sigma^\star$, deux entiers $n_1 \in \mathbb{N}$ et $n_2 \in \mathbb{N}$
Sortie : Existe-t-il une machine $\mathcal{N}$ dont la sérialisation est de taille $\leqslant n_1$ et terminant en moins de n_2 étapes élémentaires sur l'entrée w ?

2.5 Le problème de l'arrêt

Les résultats de la section précédente sont très encourageants : le modèle de calcul que nous nous sommes donné est assez expressif pour permettre la définition de machines universelles, c'est-à-dire de machines capables de simuler l'exécution d'autres machines. Demandons-nous si ces machines peuvent aussi décider des propriétés sur d'autres machine, par exemple, une machine peut-elle décider si une autre machine s'arrête sur un mot donné ? Cela revient à se demander si le problème suivant, le classique **problème de l'arrêt**, est décidable.

Arrêt : **Entrée :** Une machine $\mathcal{M}$ et un mot w
Sortie : L'exécution de la machine $\mathcal{M}$ sur l'entrée w s'arrête-t-elle ?

Théorème 2.53

Le problème Arrêt est indécidable.

Démonstration : Supposons par l'absurde que le problème Arrêt soit décidable. Il existe alors une machine `arret : string -> bool` prenant en argument une chaîne de caractères représentant un couple contenant une machine $\mathcal{M}$ et une entrée w et retournant si oui ou non la machine $\mathcal{M}$ s'arrête lorsqu'on l'exécute sur l'entrée w. Cette machine s'arrête sur toute entrée, en particulier si la chaîne passée en argument n'est pas la sérialisation d'un couple contenant la sérialisation d'une machine et d'un mot, la machine `arret` termine et renvoie `false`. Définissons alors la fonction `paradoxe : string -> bool` comme suit où la fonction `serialise_couple : string -> string -> string` prend en arguments deux chaînes de caractères w_1 et w_2 et retourne la sérialisation du couple (w_1, w_2).

```
let paradoxe (s: string) : bool =
  if arret (serialise_couple s s) then
    (while true do () done; true)
  else false
```

Notons alors `w_p` la sérialisation de la fonction `paradoxe`, c'est-à-dire la chaîne de caractères suivante.

```
"let paradoxe (s: string) : bool = if arret (serialise_couple s s) then (while true do
↪  () done; true) else false"
```

Ainsi par définition de la fonction `arret`, pour toute chaîne de caractères `w`, `arret (serialise_couple w_p w)` vaut `true` si `paradoxe` s'arrête sur `w` et `false` sinon. Considérons alors en particulier le cas où l'on choisit pour `w` la chaîne `w_p`.

- Si l'exécution de `(paradoxe wp)` termine, c'est donc que la fonction `paradoxe` a exécuté la branche **`else`** de son **`if`**, c'est donc que `arret (serialise_couple w_p w_p)` vaut `false`. C'est donc que la fonction `arret` répond `false` lorsqu'on lui demande si l'appel de la fonction `paradoxe` sur l'entrée `w_p` termine, ce qui est absurde puisqu'on a supposé que l'exécution terminait.
- Si l'exécution de `(paradoxe w_p)` ne termine pas, c'est donc que la fonction `paradoxe` a exécutée la branche **`then`** de son **`if`**, c'est donc que `arret (serialise_couple w_p w_p)` vaut `true`. C'est donc que la fonction `arret` répond `true` lorsqu'on lui demande si l'appel de la fonction `paradoxe` sur l'entrée `w_p` termine, ce qui est absurde puisqu'on a supposé que l'exécution ne terminait pas.

On conclut donc à une absurdité Absurde et le problème de l'Arrêt ne peut être décidable. □

Corollaire 2.54

Il existe des problèmes indécidables.

Exercice de cours 2.55

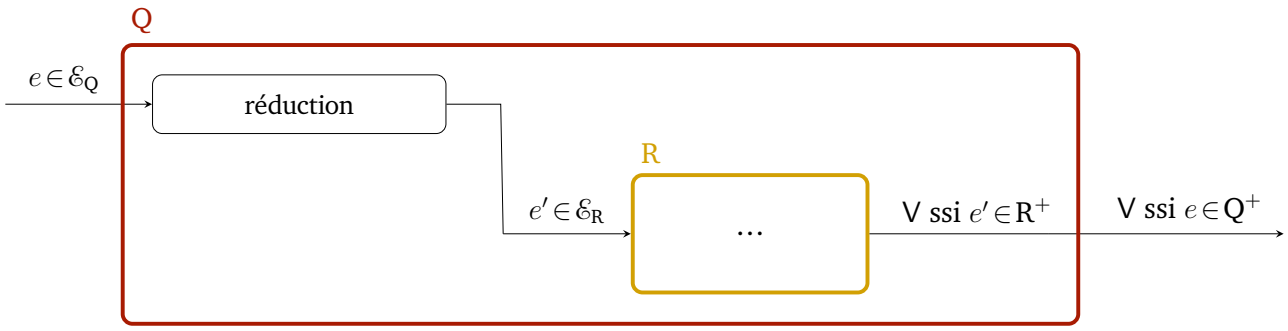
Démontrer que le problème suivant est indécidable.

Boucle : **Entrée :** Une machine $\mathcal{M}$ et un mot w
Sortie : L'exécution de la machine $\mathcal{M}$ sur l'entrée w conduit-elle à un comportement d'erreur ?

2.6 Réduction

Dans la section précédente nous avons donné un seul problème indécidable, alors que les arguments de cardinalité développés précédemment nous assurent qu'il en existe une infinité non dénombrable. On présente ici une notion qui nous permettra de déduire de l'indécidabilité du problème de l'arrêt l'indécidabilité de nombreux autres problèmes : la réduction.

Intuition. Avant de donner la définition formelle, on explique intuitivement ce qu'est une réduction d'un problème de décision Q à un problème de décision R. Il s'agit d'une manière de transformer une instance de Q en une instance de R, de sorte que la réponse de R pour cette nouvelle instance soit exactement celle de Q pour l'instance de départ. Ainsi une telle réduction donne une manière de résoudre Q en s'appuyant sur R. On représente une telle réduction avec le schéma ci-dessous.



Définition 2.56

Soient Q et R deux problèmes de décision.
*Une **réduction** du problème Q vers le problème R est une fonction totale calculable $f : \mathcal{E}_Q \to \mathcal{E}_R$ telle que :*

$$\forall w \in \mathcal{E}_Q, w \in Q^+ \Leftrightarrow f(w) \in R^+.$$

*On dit que Q **se réduit à** R, et on note $Q \preccurlyeq R$, s'il existe une fonction de réduction de Q vers R.*

La relation $Q \preccurlyeq R$ signifie que le problème R est "plus dur" à résoudre que le problème Q. En effet la connaissance d'une solution pour le problème R donne une solution pour le problème Q.

Exemple 2.57
Considérons les deux problèmes de décision suivants.

$$\textsc{Rectangle} : \begin{cases} \textbf{Entrée :} (a,b,s) \in \mathbb{N}^3 \\ \textbf{Sortie} \ \textbf{:} \ a \times b = s \, ? \end{cases} \qquad \textsc{Carré} : \begin{cases} \textbf{Entrée :} (c,s) \in \mathbb{N}^2 \\ \textbf{Sortie} \ \textbf{:} \ c \times c = s \, ? \end{cases}$$

Le problème Carré se réduit au problème Rectangle par la fonction $\begin{pmatrix} \mathbb{N}^3 & \to & \mathbb{N}^2 \\ (c,s) & \mapsto & (c,c,s) \end{pmatrix}$.

Exercice de cours 2.58

Considérons le problème $\textsc{Contient}_{\texttt{'a'}}$ défini page 9 et le problème $\textsc{Contient}_{\texttt{'a\&b'}}$ définit comme suit.

$$\textsc{Contient}_{\texttt{'a\&b'}} : \begin{cases} \textbf{Entrée :} \ s \in \Sigma^\star \text{ une chaîne de caractères} \\ \textbf{Sortie} \ \textbf{:} \ \text{La chaîne } s \text{ contient-elle le caractère } \texttt{'a'} \text{ et le caractère } \texttt{'b'} \, ? \end{cases}$$

Montrer que le problème $\textsc{Contient}_{\texttt{'a'}}$ se réduit au problème $\textsc{Contient}_{\texttt{'a\&b'}}$.

Proposition 2.59

La relation $\preccurlyeq$ définit un pré-ordre, c'est-à-dire une relation binaire réflexive et transitive.

Démonstration : Montrons que $\preccurlyeq$ est réflexive et transitive.

- Soit Q un problème de décision. La fonction totale identité $f : \mathcal{E}_Q \to \mathcal{E}_Q$ est bien calculable, et est une réduction de Q vers Q puisque $\forall w \in \mathcal{E}_Q, w \in Q \Leftrightarrow f(w) \in Q$.
- Soient Q, R et S trois problèmes de décisions tels que $Q \preccurlyeq R$ et $R \preccurlyeq S$. On note alors f_1 (resp. f_2) la fonction de réduction de Q vers R (resp. de R vers S), ainsi $\forall w \in \mathcal{E}_Q, \ w \in Q^+ \Leftrightarrow f_1(w) \in R^+$ et $\forall w \in \mathcal{E}_R, \ w \in R^+ \Leftrightarrow f_2(w) \in S^+$. On considère alors la fonction totale $f_3 = f_2 \circ f_1$. Cette fonction, de $\mathcal{E}_Q$ dans $\mathcal{E}_S$, est bien calculable en tant que composée de deux fonctions calculables. De plus $\forall w \in \mathcal{E}_Q, \ f_3(w) \in S^+ \Leftrightarrow f_2(f_1(w)) \in S^+ \Leftrightarrow f_1(w) \in R^+ \Leftrightarrow w \in Q^+$ donc f_3 est bien une réduction de Q vers S.

□

Exemple 2.60
On donne ici l'exemple de deux problèmes différents équivalents, *i.e.* se réduisant l'un à l'autre, ce qui fournit un contre-exemple à l'antisymétrie de $\preccurlyeq$.

$$\textsc{NulÀDroite} : \begin{cases} \textbf{Entrée :} (a,b) \in \mathbb{N}^2 \\ \textbf{Sortie} \ \textbf{:} \ b \text{ est-il nul} \, ? \end{cases} \qquad \textsc{NulÀGauche} : \begin{cases} \textbf{Entrée :} (a,b) \in \mathbb{N}^2 \\ \textbf{Sortie} \ \textbf{:} \ a \text{ est-il nul} \, ? \end{cases}$$

Le problème NulÀDroite se réduit au problème NulÀGauche par la fonction $\begin{pmatrix} \mathbb{N}^2 & \to & \mathbb{N}^2 \\ (x,y) & \mapsto & (y,x) \end{pmatrix}$.

De plus NulÀGauche se réduit au problème NulÀDroite par la même fonction de réduction.

Remarque 2.61

Un pré-ordre n'est donc pas un ordre. Il lui manque la propriété d'antisymétrie. On ne confondra pas un **pré-ordre** et un ordre **partiel**.

Proposition 2.62

Soient Q et R deux problèmes de décision tels que $Q \preccurlyeq R$.
Si R est décidable, alors Q l'est aussi.

Démonstration : Puisque $\mathrm{Q} \preccurlyeq \mathrm{R}$, il existe alors une fonction totale calculable $f : \mathcal{E}_\mathrm{Q} \to \mathcal{E}_\mathrm{R}$ vérifiant $\forall w \in \mathcal{E}_\mathrm{Q}$, $w \in \mathrm{Q}^+ \Leftrightarrow f(w) \in \mathrm{R}^+$. Supposons que R est décidable. Il existe alors une fonction totale calculable f_R décidant le problème R. La fonction $f_\mathrm{R} \circ f$, totale et calculable en tant que composée de fonctions totales calculables, décide alors le problème R. En effet, $\forall w \in \mathcal{E}_\mathrm{Q}, f_\mathrm{R} \circ f(w) = \mathsf{V} \Leftrightarrow f(w) \in \mathrm{R}^+ \Leftrightarrow w \in \mathrm{Q}^+$. □

Proposition 2.63

Soient Q et R deux problèmes de décision tels que $Q \preccurlyeq R$.
Si Q est indécidable, alors R l'est aussi.

Démonstration : C'est la contraposée du résultat précédant. □

Exercice de cours 2.64

Montrer que si Q et R sont deux problèmes décidables non triviaux, alors il existe une réduction de Q à R.

Le problème de la vacuité. On montre ici comment on utilise en pratique la notion de réduction pour montrer qu'un problème est indécidable. Le résultat d'indécidabilité n'est pas à retenir en tant que tel, c'est au contraire la technique de preuve qu'il faut retenir, c'est pourquoi les étapes clés ont été explicitées en italique dans la preuve qui suit.

$$\textsc{NonVide} : \begin{cases} \textbf{Entrée :} \text{ La sérialisation d'une machine } \mathcal{M} \\ \textbf{Sortie :} \text{ A-t-on } \mathcal{L}(\mathcal{M}) \neq \{\emptyset\} \text{ ?} \end{cases}$$

Proposition 2.65

Le problème NonVide est indécidable.

Démonstration : en 4 étapes.

1. *Annoncer la preuve par réduction, et expliciter le sens de la réduction.*
   Montrons que NonVide est indécidable par réduction depuis le problème Arrêt.
2. *Proposer une fonction de réduction calculable.*
   Pour $(\mathcal{M}, w)$ une instance de Arrêt, on peut construire une machine $\mathcal{M}'$ qui prend en entrée un mot w', exécute $\mathcal{M}$ sur w, puis accepte le mot w', *i.e.* renvoie V. L'existence d'une machine universelle nous assure qu'il est possible de réaliser cette construction, autrement dit que la fonction qui à $(\mathcal{M}, w)$ associe $\mathcal{M}'$ est calculable. Pour le justifier formellement, on peut considérer la fonction OCaml ci-dessous.

```
let red (s: string) : string =
  let (m, w) = deserialise_couple s in
  "fun s ->
     if s <> "^w^" then false
     else let _ = interprete "^m^" "^w^" in true
  "
```

3. *Montrer qu'il y a équivalence entre la positivité de l'instance de départ et celle après réduction.*
   Vue la construction de $\mathcal{M}'$, il y a deux cas possibles. Ou bien la machine $\mathcal{M}$ termine sur w, auquel cas $\mathcal{M}'$ accepte tous les mots w', donc $\mathcal{L}(\mathcal{M}') = \Sigma^\star \neq \emptyset$, ou bien $\mathcal{M}$ ne termine pas sur w, auquel cas $\mathcal{M}'$

bloque sur tout mot w', donc $\mathcal{L}(\mathcal{M}') = \emptyset$.

$$\begin{aligned} \mathcal{M}' \in \textsc{NonVide}^+ &\Leftrightarrow \mathcal{L}(\mathcal{M}') \neq \emptyset \\ &\Leftrightarrow \mathcal{M} \text{ termine sur } w \\ &\Leftrightarrow (\mathcal{M}, w) \in \textsc{Arrêt}^+ \end{aligned}$$

4. *Conclure*.
Ainsi $\textsc{Arrêt} \preccurlyeq \textsc{NonVide}$, or $\textsc{Arrêt}$ est indécidable, donc $\textsc{NonVide}$ est indécidable.

□

Exercice de cours 2.66

Démontrer que le problème suivant est indécidable.

$$\textsc{EstEpsilon} : \begin{cases} \textbf{Entrée :} \text{ La sérialisation d'une machine } \mathcal{M} \\ \textbf{Sortie :} \text{ A-t-on } \mathcal{L}(\mathcal{M}) = \{\varepsilon\} \text{ ?} \end{cases}$$

Remarque 2.67

Ce résultat est en fait un cas particulier du théorème de Rice qui assure que tester une propriété portant sur le langage d'une machine est indécidable dès lors que cette propriété n'est pas triviale (*i.e.* ni vérifiée par tous les langages de machines, ni vérifiée par aucun langage de machine). Cette preuve particulière, avec ce type de réduction, est donc doublement importante : elle donne l'idée de la démonstration de la preuve du théorème de Rice, et elle donne l'idée de la preuve d'indécidabilité des problèmes portant sur le langage d'une machine. Cette technique est développée dans un exercice de TD, mais on pourra retenir (après avoir fait l'exercice) le schéma suivant : à partir d'une machine $\mathcal{M}$ et d'un mot w, on crée une machine $\mathcal{M}'$ ayant pour langage, ou bien le langage vide si $\mathcal{M}$ ne s'arrête pas sur w, ou bien un langage qu'on a choisi parce qu'il a le statut inverse vis-à-vis de la propriété. Ainsi tester si $\mathcal{M}$ s'arrête sur w revient à tester si $\mathcal{L}(\mathcal{M}')$ vérifie la propriété.

3 Classes P et NP

Dans la section précédente nous nous sommes donnés un modèle de calcul qui nous a permis de classer les problèmes (ou de manière équivalente les langages) en deux catégories : les problèmes décidables et les problèmes indécidables. De plus la définition de ce modèle de calcul permet de parler d'une **étape** de calcul, et ainsi de compter le nombres d'étapes lors de l'exécution d'une machine. Cette notion coïncide avec la notion de complexité d'un algorithme déjà définie dans les cours de première année. Nous aimerions cette année caractériser la "complexité" d'un problème plutôt que celle des algorithmes qui le résolvent. Aussi dans cette section nous allons définir deux nouvelles classes de problèmes : la classe P et la classe NP et montrer comment la notion de réduction permet d'assurer la relative difficulté de résolution d'un problème.

3.1 Complexité d'une machine

Taille des entrées. Le nombre d'opérations élémentaires que nécessite une machine pour résoudre un problème dépend le plus souvent de l'instance du problème considérée, c'est-à-dire de l'entrée sur laquelle s'exécute la machine, et en particulier de la taille de cette entrée. Puisqu'on a choisi un modèle de calcul où les machines prennent en entrée des chaînes de caractères, la taille d'une entrée peut être définie comme sa longueur en tant que chaîne de caractères. Si w est une chaîne, on note $|w|$ sa longueur.

Remarque 3.1

Dans la suite on décrira les complexités en donnant leur comportement asymptotique au moyen des notations de Landau (O, Ω, et Θ), comme on l'a déjà fait pour la complexité des algorithmes. Dans cette optique, connaître la longueur précise de la chaîne importe peu, ainsi on continuera de dire qu'un tableau à n éléments est de taille n, même si sa sérialisation est peut-être une chaîne de $2n+1$ caractères. Néanmoins le codage des données peut avoir son importance, on pourra par exemple réfléchir à la variation de complexité d'un algorithme qui opère sur un entier encodé en base 2 ou sur un entier encodé en base 1.

Notation 3.2

Étant donné une machine $\mathcal{M}$ et une entrée $w \in \Sigma^*$, on note :

- $c^{\mathcal{M}}(w) = \begin{cases} +\infty \text{ si l'exécution de } \mathcal{M} \text{ sur } w \text{ ne termine pas,} \\ \text{le nombre d'opérations élémentaires effectuées lors de cette exécution sinon ;} \end{cases}$
- $c_n^{\mathcal{M}}$ *le nombre maximum d'opérations élémentaires qu'effectue $\mathcal{M}$ sur une entrée de taille $n \in \mathbb{N}$, i.e.* $\forall n \in \mathbb{N},\ c_n^{\mathcal{M}} = \max\{c^{\mathcal{M}}(w) \mid |w| = n\} \in \overline{\mathbb{N}}$.

Remarque 3.3

Pour tout $n \in \mathbb{N}$ et pour toute machine $\mathcal{M}$, si $c_n^{\mathcal{M}}$ est fini alors pour tout mot $w \in \Sigma^\star$ tel que $|w| = n$ l'exécution de $\mathcal{M}$ sur w termine, autrement dit la machine $\mathcal{M}$ termine sur toute entrée de taille n.

3.2 Classe P

Définition 3.4

On dit d'une machine $\mathcal{M}$ qu'elle est de ***complexité polynomiale*** *s'il existe un entier $p \in \mathbb{N}$ tel que pour tout $n \in \mathbb{N}$, $c_n^{\mathcal{M}}$ est fini et $c_n^{\mathcal{M}} = \mathcal{O}(n^p)$.*

Lemme 3.5

Si une machine $\mathcal{M}$ est de complexité polynomiale, il existe un polynôme B, croissant sur $\mathbb{N}$ tel que $\forall n \in \mathbb{N}, c_n^{\mathcal{M}} \leqslant B(n)$, autrement dit tel que pour tout $w \in \Sigma^\star$, l'exécution de $\mathcal{M}$ sur w conduit à un nombre d'opérations élémentaires majoré par $B(|w|)$.

Exercice de cours 3.6

Démontrer ce résultat.

Définition 3.7

On dit d'une fonction (partielle ou non) qu'elle est ***calculable en complexité polynomiale*** *s'il existe une machine de complexité polynomiale qui la calcule.*

Exemple 3.8

- La fonction d'incrémentation d'un entier codé en base 2 est polynomiale.
- La fonction prenant en argument un graphe G, deux sommets u et v et testant l'accessibilité de v depuis u est polynomiale.

Exercice de cours 3.9

Justifier ces deux exemples en fournissant des algorithmes de complexité polynomiale.

Proposition 3.10

La composée de deux fonctions totales calculables en complexité polynomiale est une fonction totale calculable en complexité polynomiale.

Démonstration : Soit $f_1 : \Sigma^\star \to \Sigma^\star$ calculée par une machine `mac1` de complexité polynomiale et $f_2 : \Sigma^\star \to \Sigma^\star$ calculée par une machine `mac2` de complexité polynomiale. Alors la machine : `mac = fun s -> mac2 (mac1 s)` calcule $f_2 \circ f_1$, et on va montrer qu'elle est de complexité polynomiale. Puisque `mac1` et `mac2` sont de complexité polynomiale, il existe A_1 et A_2 deux polynômes croissants sur $\mathbb{N}$, tels que $\forall w \in \Sigma^\star, c^{\texttt{mac1}}(w) \leqslant A_1(|w|)$ et $c^{\texttt{mac2}}(w) \leqslant A_2(|w|)$. Cela assure en particulier que la machine `mac` s'arrête sur toute entrée. Par ailleurs on remarque que la taille de `(mac1 s)` est majorée par $A_1(t)$ où t est la taille de s, en effet la construction d'une chaîne de m caractères nécessite au moins m opérations élémentaires. Alors le nombre d'opérations élémentaires effectuées par `mac`, sur une entrée w, est majoré par $A_2(|\texttt{mac1}(w)|)+A_1(|w|)+1 \leqslant A_2(A_1(|w|))+A_1(|w|)+1$. Le polynôme $B = A_2 \circ A_1 + A_1 + 1$ permet donc d'affirmer que `mac` est calculable en complexité polynomiale. □

Exercice de cours 3.11

La preuve précédente fonctionne-t-elle si on s'intéresse aux fonctions calculables en complexité quadratique (*i.e.* en $\mathcal{O}(n^2)$) ?

Définition 3.12

On dit d'un problème de décision qu'il est ***décidable en temps polynomial*** *s'il existe une machine de complexité polynomiale qui le décide. On définit* ***la classe P*** *comme étant l'ensemble des problèmes de décision décidables en temps polynomial.*

Exemple 3.13

- Soit $\mathcal{A}$ un automate déterministe, le problème de l'appartenance d'un mot au langage $\mathcal{L}(\mathcal{A})$ dans P, il en est de même lorsque $\mathcal{A}$ n'est pas déterministe.
- Le problème SATISFAIT de savoir si un environnement donné satisfait une formule donnée est dans P.

$$\text{SATISFAIT} : \begin{cases} \textbf{Entrée :} G \text{ une formule de la logique propositionnelle sur } \mathcal{Q},\ \rho \in \mathbb{B}^{\mathcal{Q}} \\ \textbf{Sortie :} \text{A-t-on } [\![G]\!]^\rho = \mathsf{V} \text{ ?} \end{cases}$$

Exercice de cours 3.14

Justifier ces deux exemples en fournissant des algorithmes de complexité polynomiale permettant de résoudre ces problèmes.

Remarque 3.15

On dit par extension qu'un langage L est dans la classe P dès lors que le problème APPARTIENT_L est dans la classe P, ce qu'on note $L \in \mathsf{P}$.

Proposition 3.16

Pour L_1 et L_2 deux langages de la classe P :

- $\emptyset \in P$
- $\Sigma^\star \in P$
- $L_1 \cup L_2 \in P$
- $L_1 \cap L_2 \in P$
- $\overline{L_1} \in P$
- $L_1 \cdot L_2 \in P$
- $L_1^\star \in P$

Exercice de cours 3.17

Démontrer la proposition précédente. $L_1^\star \in \mathsf{P}$ est plus difficile à démontrer que les autres.

3.3 Classe NP

Attention : Si P signifie polynomial, NP ne signifie pas non polynomial.

Définition 3.18

*Soit Q un problème de décision. On dit que Q est dans **la classe NP** dès lors qu'il existe :*
- *un polynôme A,*
- *$\mathcal{C} \subseteq \Sigma^\star$ un langage de la classe P,*
- *Vérif un problème de décision sur $\mathcal{E}_Q \times \mathcal{C}$ de la classe P,*

tels que

$$\forall w \in \Sigma^*,\ w \in Q^+ \Leftrightarrow \exists c \in \mathcal{C}, |c| \leqslant A(|w|) \text{ et } (w,c) \in \text{Vérif}^+. \qquad (\bigstar)$$

*L'ensemble $\mathcal{C}$ est alors appelé ensemble des **certificats** et le problème Vérif est alors appelé **problème de vérification** associé.*

Un problème de la classe NP est donc un langage pour lequel on peut vérifier en temps polynomial, à l'aide d'un certificat obtenu par ailleurs, qu'une instance est positive. On comprend ici que les instances positives et négatives ont un rôle dissymétrique : une instance est négative si et seulement s'il n'existe pas de certificat assez petit qui la justifie.

Remarque 3.19

Une définition équivalente n'utilisant pas l'ensemble des certificats est la suivante : un problème de décision Q est dans NP si et seulement s'il existe un exposant $p \in \mathbb{N}$ et un problème Vérif $\in$ P, tel que :

$$\forall w \in \Sigma^*,\ w \in \mathrm{Q}^+ \Leftrightarrow \exists c \in \Sigma^\star, |c| \leqslant |w|^p \text{ et } (w,c) \in \text{Vérif}^+.$$

Le problème de la vérification que u est un certificat valide est repoussé vers Vérif. Cette définition, quoique plus courte, n'est pas celle qu'on attend dans les exercices, car expliciter l'ensemble des certificats simplifie souvent la définition du problème Verif.

Remarque 3.20

Les certificats ont vocation à être des mots qui décrivent simplement une justification de la positivité de l'instance, notamment des témoins d'existence pour tous les problèmes de la forme "Existe-t-il … ? ". Dans ce cadre, l'ensemble de certificats est souvent naturellement dans P. La raison pour laquelle cette condition apparaît dans la définition est d'éviter les détournements du type "prenons $\mathcal{C} = \mathrm{Q}^+$ et Vérif qui consiste à décider l'égalité entre l'entrée et le certificat, ainsi n'importe quel problème est dans NP ". À l'inverse la condition Vérif $\in$ P est vraiment centrale, et on prendra soin de justifier pourquoi les problèmes de vérification proposés sont dans P.

Remarque 3.21

Une définition plus "classique" mais hors programme de la classe NP passe par un changement de modèle de calcul. De la même manière que les automates non déterministes sont une variante des automates déterministes permettant l'exécution "simultanée" de plusieurs (un nombre fini tout de même) exécutions, on peut définir un modèle de calcul **non déterministe** des programmes OCaml dont l'exécution conduirait à plusieurs exécutions simultanées. De la même manière que dans les automates, on pourrait alors définir l'acceptation d'un mot par une telle machine comme l'acceptation du mot par **une** des exécutions. Le N de NP ne vient pas de Non polynomiale mais bien de Non déterministe polynomiale en ce sens que c'est l'ensemble des mots reconnus par un programme OCaml s'exécutant en complexité polynomiale, mais de manière non déterministe.

Le problème SubsetSum. On montre ici comment on utilise en pratique la définition de la classe NP. Le résultat sur SubsetSum n'est pas à retenir en tant que tel, c'est au contraire la technique de preuve qu'il faut retenir, c'est pourquoi les étapes clés ont été explicitées en italique dans la preuve qui suit.

$$\text{SubsetSum} : \begin{cases} \textbf{Entrée :} \text{ Un entier } n \in \mathbb{N}, \text{ une suite finie } (w_i)_{i\in[\![1,n]\!]} \in \mathbb{N}^n, \text{ un entier } W \\ \textbf{Sortie :} \text{ Existe-t-il } I \subseteq [\![1,n]\!] \text{ tel que } \sum_{i\in I} w_i = W \text{ ?} \end{cases}$$

Proposition 3.22

Le problème SubsetSum est dans NP.

Démonstration : en 4 étapes, dont les deux dernières commutent.

0. *On trouve, sans forcément l'écrire, comment certifier qu'une instance est positive à l'aide d'un certificat de taille polynomiale.*
   Pour certifier qu'une instance à n entiers est positive, il suffit de donner un sous-ensemble I de $[\![1,n]\!]$ qui convient♣. Un tel sous-ensemble peut être encodé par n booléens, soit par un mot de taille n, or l'instance est de taille $\geqslant n$ ♡, donc le certificat envisagé est bien polynomial en la taille de l'instance.
1. *On en déduit l'ensemble de certificats* ♠.
   On pose $\mathcal{C} = \bigcup_{n\in\mathbb{N}} \mathcal{C}_n$ où $\forall n \in \mathbb{N}$, $\mathcal{C}_n = \{0,1\}^n$.
2. *On fixe le problème de vérification associé.*
   On considère alors le problème de vérification suivant.

$$\text{Vérif} : \begin{cases} \textbf{Entrée :} \text{ Une instance de SubsetSum } (n,w,W), \text{ un certificat } c \in \mathcal{C} \\ \textbf{Sortie :} \text{ A-t-on } \sum\limits_{\substack{i\in[\![1,n]\!] \\ c_i=1}} w_i = W \text{ ?} \end{cases}$$

3. *On justifie que le problème de vérification est dans P.*
   Étant donné une instance (n,w,W) de SubsetSum et $c \in \mathcal{C}$, on peut vérifier en $O(n)$ que c encode bien un sous-ensemble I de $[\![1,n]\!]$. On peut alors sommer les valeurs w_i pour $i \in I$, et vérifier que la valeur obtenue est bien W, et ce encore en $O(n)$. Ainsi le problème Vérif est dans P.
4. *On justifie que l'équivalence* $(\bigstar)$ *est satisfaite.* ♢.
   Soit $e = (n,w,W)$ une entrée de SubsetSum.

$$\begin{aligned} (n,w,W) \in \text{SubsetSum}^+ &\Leftrightarrow \exists I \subseteq [\![1,n]\!], \sum_{i\in I} w_i = W \\ &\Leftrightarrow \exists c \in \mathcal{C}_n, \sum_{\substack{i\in[\![1,n]\!] \\ c_i=1}} w_i = W \\ &\Leftrightarrow \exists c \in \mathcal{C}_n, ((n,w,W),c) \in \text{Vérif}^+ \\ &\Leftrightarrow \exists c \in \mathcal{C}, |c| \leqslant n \text{ et } (e,c) \in \text{Vérif}^+ \\ &\Leftrightarrow \exists c \in \mathcal{C}, |c| \leqslant |e| \text{ et } (e,c) \in \text{Vérif}^+ \end{aligned}$$

Finalement SubsetSum est bien un problème de NP. □

♣. Lorsque le problème est formulé comme un problème d'existence, trouver le certificat est assez facile, il faut néanmoins s'assurer que sa taille est polynomiale.

♡. On prendra garde à ne pas conclure trop vite dans les cas où n est une donnée d'entrée mais que l'entrée n'est que de taille en $\log(n)$.

♠. **Attention :** cet ensemble doit être le même pour toutes les instances, pas seulement celles de taille n, qui d'ailleurs n'est pas fixé ici.

♢. Le polynôme A utilisé peut rester implicite. Ici le polynôme $A(X) = X$ conviendrait.

Exercice de cours 3.23

Considérons le problème suivant.

$$\textsc{Rendu}\begin{cases}\textbf{Entrée :} \text{ Un entier } n\in\mathbb{N},\ (p_1,p_2,\dots,p_n)\in\mathbb{N}^n \text{ avec } p_1=1, \text{ un entier } M\in\mathbb{N}, \text{ un entier } K\in\mathbb{N}\\ \textbf{Sortie} \ : \text{Existe-t-il } (c_1,c_2,\dots,c_n)\in\mathbb{N}^n \text{ tels que } \sum_{i=1}^n c_ip_i=M \text{ et } \sum_{i=1}^n c_i\leqslant K\,?\end{cases}$$

Démontrer que ce problème est dans NP.

Proposition 3.24

$P\subseteq NP$

Démonstration : *L'idée est qu'il n'y a, ici, pas besoin de certificat pour vérifier qu'une instance du problème est positive, puisqu'on peut simplement résoudre le problème pour cette instance (en complexité polynomiale).*

Soit un problème $\mathsf{Q}\in\mathsf{P}$. On pose $\mathcal{C}=\{\varepsilon\}$, et on considère alors le problème de vérification suivant.

$$\textsc{Vérif} : \begin{cases}\textbf{Entrée :} (w,c)\in\mathcal{E}_\mathsf{Q}\times\mathcal{C}\\ \textbf{Sortie} \ : w\in\mathsf{Q}^+\end{cases}$$

Ainsi $\forall w\in\mathcal{E}_\mathsf{Q}, w\in\mathsf{Q}^+\Leftrightarrow(w,\varepsilon)\in\textsc{Vérif}^+$. En considérant de plus le polynôme $A=1$, on en déduit que $\forall w\in\mathcal{E}_a, w\in\mathsf{Q}^+\Leftrightarrow\exists c\in\mathcal{C},|c|\leqslant A(|w|)$ et $(w,c)\in\textsc{Vérif}^+$. De plus, puisque $\mathsf{Q}\in\mathsf{P}$, il existe une machine de complexité polynomiale $\mathcal{M}$ décidant Q. La machine $\mathcal{M}'$ qui prend en entrée un couple $(w,c)\in\mathcal{E}_\mathsf{Q}\times\mathcal{C}$, ignore c, puis se comporte comme $\mathcal{M}$ sur w résout alors le problème Vérif et est de complexité polynomiale. Ainsi $\textsc{Vérif}\in P$. L'existence de tels $\mathcal{C}$, A et Vérif justifie que $\mathsf{Q}\in\mathsf{NP}$. □

Remarque 3.25

La fameuse question $\mathsf{P}\stackrel{?}{=}\mathsf{NP}$ est le problème de savoir si l'inclusion précédente est stricte ou non. Ce problème n'a pas encore de réponse et est un des problèmes centraux en informatique théorique.

Proposition 3.26

Sat $\in$ NP.

Démonstration : Rappelons le problème Sat.

$$\textsc{Sat} : \begin{cases}\textbf{Entrée} \ : \ n\in\mathbb{N},\ G \text{ une formule de la logique propositionnelle sur } \{x_1,x_2\dots x_n\}\,?\\ \textbf{Sortie} \ : \ G \text{ est-elle satisfiable ?}\end{cases}$$

Soit Vérif le problème de vérification suivant.

$$\textsc{Vérif} : \begin{cases}\textbf{Entrée} \ : \ (n,G) \text{ une instance de Sat, un env. prop. } \rho \text{ défini sur } \mathsf{vars}(G)\\ \textbf{Sortie} \ : \ \text{A-t-on } [\![G]\!]^\rho=\mathsf{V}\,?\end{cases}$$

En fait le problème Vérif introduit ici est le problème Satisfait dont on a montré qu'il est dans P dans l'exercice de cours 3.14.

Pour toute formule propositionnelle G :

$$G\in\textsc{Sat}\Leftrightarrow\exists\rho\in\mathbb{B}^{\mathsf{vars}(G)},[\![G]\!]^\rho=\mathsf{V}.$$

Or un environnement propositionnel de $\mathbb{B}^{\mathsf{vars}(G)}$ peut être représenté au moyen d'une liste de $|\mathsf{vars}(G)|\leqslant|G|$ couples. Ainsi ρ peut être représenté au moyen d'une chaîne de caractères de taille polynomiale en $|G|$. □

3.4 NP-difficulté

Montrer qu'un problème est dans la classe NP n'apporte toutefois aucune information sur la borne inférieure de complexité de résolution de ce problème. Un problème pouvant être résolu en complexité constante, par exemple, est dans la classe NP. Pour se convaincre qu'un problème est difficile à résoudre algorithmiquement on préférera donc montrer qu'il est au moins aussi difficile à résoudre que le plus dur des problèmes de la classe NP, ou encore qu'il est au moins aussi dur à résoudre que tous les problèmes de la NP. Cette comparaison de la difficulté des problèmes se fait grâce à la notion de réduction polynomiale. ♣

Définition 3.27

Soient deux problèmes de décision Q et R, sur des ensembles d'entrées respectifs $\mathcal{E}_Q$ et $\mathcal{E}_R$.
*Une **réduction polynomiale** du problème Q vers le problème R est une fonction totale calculable en complexité polynomiale $f : \mathcal{E}_Q \to \mathcal{E}_R$ telle que :*

$$\forall w \in \mathcal{E}_Q, w \in Q^+ \Leftrightarrow f(w) \in R^+.$$

On dit que Q se réduit en complexité polynomiale à R, et on note $Q \preccurlyeq_P R$ lorsqu'il existe une telle réduction de Q vers R.

Proposition 3.28

La relation $\preccurlyeq_P$ définit un pré-ordre, c'est-à-dire une relation réflexive et transitive.

Démonstration : • Soit un problème Q, considérons alors la fonction totale identité (qui est calculable en temps linéaire donc polynomial) $f : \mathcal{E}_Q \to \mathcal{E}_Q$, alors $\forall w \in \mathcal{E}_Q, w \in Q \Leftrightarrow f(w) \in Q$.

- Si Q, R et S sont trois problèmes tels que $Q \preccurlyeq_P R$ et $R \preccurlyeq_P S$, il existe alors une fonction totale calculable $f_1 : \mathcal{E}_Q \to \mathcal{E}_R$ telle que $\forall w \in \mathcal{E}_Q, w \in Q \Leftrightarrow f_1(w) \in R$ et une fonction totale calculable $f_2 : \mathcal{E}_R \to \mathcal{E}_S$ telle que $\forall w \in \mathcal{E}_R, w \in R \Leftrightarrow f_2(w) \in S$. Considérons alors la fonction totale $f_3 = f_2 \circ f_1$. Cette fonction f_3 est calculable en complexité polynomiale, de $\mathcal{E}_Q$ dans $\mathcal{E}_S$. De plus pour $w \in \mathcal{E}_Q$ elle vérifie :
  - si $w \in Q^+$ alors $f_1(w) \in R^+$ et donc $f_2(f_1(w)) \in S^+$ soit $f_3(w) \in S^+$,
  - et réciproquement si $f_2(f_1(w)) \in S^+$ alors $f_1(w) \in R^+$ et donc $w \in Q^+$.
  Ainsi f_3 est une réduction polynomiale de Q vers S, donc $Q \preccurlyeq_P S$. D'où la transitivité.

□

Remarque 3.29

La relation $\preccurlyeq_P$ n'est pas antisymétrique. Le contre-exemple à l'antisymétrie de la réduction simple donné page 18 convient aussi pour montrer l'antisymétrie de la réduction polynomiale.

Remarque 3.30

On remarque ici que la stabilité par composition des fonctions totales calculables en complexité polynomiale est un élément primordial de la preuve. C'est pour cette raison que l'on ne s'intéresse pas aux classes de complexité $O(n)$, $O(n^2)$, ... qui perdent la stabilité par composition.

On rappelle que si Q se réduit au problème R alors Q est plus simple à résoudre que le problème R. Cette remarque vaut pour la réduction polynomiale tant que pour la réduction vue à la section précédente.

♣. On peut remarquer qu'entre deux problèmes décidables il existe toujours une réduction (*Cf.* 2.64), ainsi la notion de réduction vue précédemment ne permet pas de comparer les problèmes de la classe NP, elle les voit comme tous équivalents. C'est pourquoi on utilise ici la notion de réduction polynomiale qui est un peu plus forte.

Exercice de cours 3.31

Soient Q et R deux problèmes de décision tels que $\mathrm{Q} \preccurlyeq_P \mathrm{R}$.
Montrer que si $\mathrm{R} \in \mathsf{P}$ alors $\mathrm{Q} \in \mathsf{P}$.

Définition 3.32

*Un problème de décision Q est dit **NP-dur** ou **NP-difficile** dès lors que tout problème de NP se réduit en complexité polynomiale au problème Q. Autrement dit ce problème est plus dur que n'importe quel problème de NP.*

Remarque 3.33

On peut penser aux problèmes NP-complets comme à des problèmes qu'on ne peut pas résoudre en temps polynomial. Bien qu'il n'existe pas de preuve de cette impossibilité, il est hautement improbable de trouver un algorithme polynomial décidant un problème NP-complet, car cela donnerait un algorithme polynomial permettant de résoudre n'importe quel problème NP, et donc l'inclusion $\mathsf{NP} \subseteq \mathsf{P}$, alors que la recherche n'a pas démontré ce résultat malgré un intérêt marqué pour la question depuis plusieurs décennies…

Proposition 3.34

Soient Q et R deux problèmes de décision tels que $\mathrm{Q} \preccurlyeq_P \mathrm{R}$.
Si Q est NP-dur, R est NP-dur.

Démonstration : On a par définition $\forall \mathrm{S} \in \mathsf{NP}, \mathrm{S} \preccurlyeq_P \mathrm{Q}$ et donc par transitivité $\forall \mathrm{S} \in \mathsf{NP}, \mathrm{S} \preccurlyeq_P \mathrm{R}$. □

Proposition 3.35

Soient Q et R deux problèmes de décision tels que $\mathrm{Q} \preccurlyeq_P \mathrm{R}$.
Si R est dans NP, alors Q est aussi dans NP.

Démonstration : Supposons que R est dans NP. Par définition, il existe alors B un polynôme, $\mathcal{C} \subseteq \Sigma^\star$ un langage de P,$\textsc{Vérif}_\mathrm{R}$ un problème de décision sur $\mathcal{E}_\mathrm{R} \times \mathcal{C}$ de la classe P tels que

$$\forall w \in \mathcal{E}_\mathrm{R},\ w \in \mathrm{R}^+ \Leftrightarrow \exists c \in \mathcal{C}, |c| \leqslant B(|w|) \text{ et } (w,c) \in \textsc{Vérif}_\mathrm{R}^+ \tag{1}$$

Puisque $\mathrm{Q} \preccurlyeq_P \mathrm{R}$, il existe par définition $\varphi : \mathcal{E}_\mathrm{Q} \to \mathcal{E}_\mathrm{R}$ calculable en complexité polynomiale telle que

$$\forall e \in \mathcal{E}_\mathrm{Q},\ e \in \mathrm{Q}^+ \Leftrightarrow \varphi(e) \in \mathrm{R}^+ \tag{2}$$

Comme φ est de complexité polynomiale, on peut borner la taille de sa sortie par un polynôme en la taille de l'entrée, ainsi il existe A un polynôme croissant sur $\mathbb{N}$ tel que

$$\forall e \in \mathcal{E}_\mathrm{Q},\ |\varphi(e)| \leqslant A(|e|) \tag{3}$$

On pose alors :

- $\mathcal{C}' = \mathcal{C} \cdot \{\#\}^\star$ (où $\#$ est un nouveau caractère, qui n'apparaît pas dans $\mathcal{C}$), On note de plus $\pi : \mathcal{C} \to \mathcal{C}'$ la fonction qui supprime les caractères $\#$ en fin de mot. Cette fonction est de complexité polynomiale.
- $\textsc{Vérif}_\mathrm{Q} = \begin{cases} \textbf{Entrée :} e \in \mathcal{E}_\mathrm{Q},\ c' \in \mathcal{C} \\ \textbf{Sortie :} \text{ V si et seulement si } |c'| = B(\varphi(e)) \text{ et } (\varphi(e), \pi(c')) \in \textsc{Vérif}_\mathrm{R}^+ \end{cases}$
- $D = B \circ A$

Puisque $\mathcal{C} \in \mathsf{P}$, on a aussi $\mathcal{C}' \in \mathsf{P}$. En effet pour tester si un mot c' est dans $\mathcal{C}'$, il suffit de calculer $c = \pi(c')$ puis de tester si $c \in \mathcal{C}'$. Puisque $\textsc{Vérif}_\mathrm{R} \in \mathsf{P}$, et que φ et π sont de complexité polynomiale, le problème $\textsc{Vérif}_\mathrm{Q}$ est aussi décidable en temps polynomial.

Enfin, pour $e \in \mathcal{E}_Q$ on a

$$\begin{aligned}
e \in Q^+ &\Leftrightarrow \varphi(e) \in R^+ && \text{par (2)}\\
&\Leftrightarrow \exists c \in \mathcal{C}, |c| \leqslant B(|\varphi(e)|) \text{ et } (\varphi(e), c) \in \text{VÉRIF}_R^+ && \text{par (1)}\\
&\Rightarrow \exists c' \in \mathcal{C}', |c'| = B(|\varphi(e)|) \text{ et } (\varphi(e), \pi(c')) \in \text{VÉRIF}_R^+ && \text{en posant } c' = c \cdot \#^p \text{ où } p = B(|\varphi(e)|) - |c| \geqslant 0\\
&\Leftrightarrow \exists c' \in \mathcal{C}', |c'| = B(|\varphi(e)|) \text{ et } (e, c') \in \text{VÉRIF}_Q^+ && \text{par définition de VÉRIF}_Q^+\\
&\Rightarrow \exists c' \in \mathcal{C}', |c'| \leqslant D(|e|) \text{ et } (e, c') \in \text{VÉRIF}_Q^+ && \text{car } B(|\varphi(e)|) \leqslant B(A(|e|)) \text{ par (3) et croissance de } B
\end{aligned}$$

Réciproquement, s'il existe $c' \in \mathcal{C}'$ tel que $|c'| \leqslant D(|e|)$ et $(e, c') \in \text{VÉRIF}_Q^+$, la définition de VÉRIF_Q^+ assure que $(\varphi(e), \pi(c')) \in \text{VÉRIF}_R^+$ et $|c'| = B(|\varphi(e)|)$, donc $|\pi(c')| \leqslant |c'| \leqslant B(|\varphi(e)|) \leqslant B(A(|e|)) = D(|e|)$. Ainsi en posant $c = \pi(c')$ on a bien $|c| \leqslant D(|e|)$ et $(\varphi(e), c) \in \text{VÉRIF}_R^+$, et donc $\varphi(e) \in R^+$ par (1) et donc $e \in Q^+$ par (2).
Ceci démontre qu'on a bien l'équivalence

$$\forall e \in \mathcal{E}_Q,\ e \in Q^+ \Leftrightarrow \exists c' \in \mathcal{C}', |c'| \leqslant D(|e|) \text{ et } (e, c') \in \text{VÉRIF}_Q^+$$

D'où $Q \in NP$ □

Théorème 3.36 (*Cook-Levin*)

SAT est NP-difficile.

Démonstration : Admis. La démonstration de ce résultat est usuellement assez technique. Le choix du modèle de calcul de ce chapitre (les programmes OCaml) rend la démonstration trop technique. □

Exercice de cours 3.37

On considère un problème d'affectation de salles pour des cours : on doit planifier n cours (numérotés de 1 à n) dans m salles (numérotées de 1 à m). Pour chaque cours $i \in [\![1, n]\!]$, on dispose d'un ensemble $S_i \in \mathcal{P}([\![1, m]\!])$ de salles dans lesquelles le cours i peut avoir lieu. On cherche alors à savoir s'il existe une attribution des salles qui permette de faire chaque cours dans une salle différente.
Formaliser le problème de décision considéré ici. Montrer qu'il est dans la classe NP. Montrer qu'il se réduit au problème SAT sans fournir de réduction (grâce au Théorème 3.36). Redémontrer qu'il se réduit au problème SAT, cette fois en construisant une réduction polynomiale vers SAT.

Remarque 3.38

Dans la suite, on montrera qu'un problème Q est NP-difficile uniquement par réduction polynomiale d'un problème déjà connu comme étant NP-difficile à ce problème Q.

Commençons par montrer que deux variantes "plus simples" du problème SAT — le problème 3SAT et le problème n-CNF-SAT — sont elles aussi NP-difficiles. Ces résultats nous permettront alors d'avoir le choix lorsqu'on souhaitera faire des réductions : partir de SAT, de 3SAT, ou de n-CNF-SAT.

Le problème 3SAT. On dit qu'une formule est une n-CNF dès lors qu'elle est sous forme normale conjonctive (conjonction de disjonctions de littéraux) et que chacune de ses clauses disjonctives contient au plus n littéraux. On définit alors, pour tout $n \in \mathbb{N}^\star$, le problème n-CNF-SAT comme étant :

$$n\text{-CNF-SAT} : \begin{cases} \textbf{Entrée :} \text{ Une } n\text{-CNF } G \\ \textbf{Sortie :} \ G \text{ est-elle satisfiable ?} \end{cases}$$

On considère alors le problème 3SAT comme étant le problème 3-CNF-SAT.

Théorème 3.39

3Sat est NP-difficile.

Démonstration : Exhibons une réduction polynomiale du problème Sat au problème 3Sat.

Soit une formule G de la logique propositionnelle sur l'ensemble de variables propositionnelles $\mathcal{Q}$. Soit $\mathcal{S}$ l'ensemble de ses sous-formules. On considère une nouvelle variable propositionnelle $p_H \notin \mathcal{Q}$ pour chaque sous-formule $H \in \mathcal{S}$. On note alors $\tilde{\mathcal{Q}} = \mathcal{Q} \cup \bigcup_{H\in\mathcal{S}} p_H$. De plus, à chaque sous-formule $H \in \mathcal{S}$ on associe G_H, une formule de la logique propositionnelle sur $\tilde{\mathcal{Q}}$ définie comme suit♣.

- $\top$ si $H = \top$
- $\bot$ si $H = \bot$
- p si $H = p \in \mathcal{Q}$
- $\neg p_{H_1}$ si $H = \neg H_1$
- $p_{H_1} \odot p_{H_2}$ si $H = H_1 \odot H_2$ avec $\odot \in \{\wedge, \vee, \to, \leftrightarrow\}$

On remarque que pour tout $H \in \mathcal{S}$, G_H a au plus 2 variables, la formule $R^0_H \overset{\text{déf}}{=} (p_H \leftrightarrow G_H)$ a donc au plus 3 variables, ainsi R^0_H est équivalente à une formule R_H sous forme 3-CNF comportant au plus 8 clauses disjonctives.

Exercice de cours 3.40

Démontrer cette affirmation en s'appuyant sur le cours de logique propositionnelle de première année.

Considérons alors la formule $R \overset{\text{déf}}{=} \bigwedge_{H\in\mathcal{S}} R_H$ (elle est équivalente à la formule $\bigwedge_{H\in\mathcal{S}} R^0_H$).

Exercice de cours 3.41

Donner la formule R obtenue dans le cas où $G = p \leftrightarrow (\neg q \to (p \wedge r))$.

Lemme 3.42

Pour tout $\rho \in \mathcal{Q} \to \mathbb{B}$, il existe $\mu \in \tilde{\mathcal{Q}}$ tel que $\mu_{|\mathcal{Q}} = \rho$ et $[\![R]\!]^\mu = V$.
Autrement dit, pour tout environnement propositionnel ρ sur l'ensemble de variables $\mathcal{Q}$, il existe un environnement propositionnel μ qui étend ρ aux nouvelles variables et qui satisfait R.

Démonstration : Soit $\rho \in \mathcal{Q} \to \mathbb{B}$. On considère l'environnement propositionnel suivant.

$$\mu : \begin{cases} \tilde{\mathcal{Q}} & \to \mathbb{B} \\ x & \mapsto \begin{cases} \rho(x) & \text{si } x \in \mathcal{Q} \\ [\![H]\!]^\rho & \text{sinon si } x = p_H \end{cases} \end{cases}$$

Un tel μ convient. Par construction $\mu_{|\mathcal{Q}} = \rho$. Montrons de plus que $[\![R]\!]^\mu = \mathsf{V}$.
Soit donc $H \in \mathcal{S}$, montrons que $[\![R_H]\!]^\mu = [\![R^0_H]\!]^\mu = \mathsf{V}$ par disjonction de cas sur H.

- Si $H = \top$, alors $R^0_H = p_H \leftrightarrow \top$, or $[\![p_H]\!]^\mu = \mu(p_H) = [\![H]\!]^\rho = [\![\top]\!]^\rho = \mathsf{V} = [\![\top]\!]^\mu$.
  Finalement $[\![p_H]\!]^\mu = [\![\top]\!]^\mu$ donc $[\![R^0_H]\!]^\mu = \mathsf{V}$.
- Si $H = \bot$, alors $R^0_H = p_H \leftrightarrow \bot$, or $[\![p_H]\!]^\mu = \mu(p_H) = [\![H]\!]^\rho = [\![\bot]\!]^\rho = \mathsf{F} = [\![\bot]\!]^\mu$.
  Finalement $[\![p_H]\!]^\mu = [\![\bot]\!]^\mu$ donc $[\![R^0_H]\!]^\mu = \mathsf{V}$.
- Si $H = p \in \mathcal{Q}$, alors $R^0_H = p_H \leftrightarrow p$, or $[\![p_H]\!]^\mu = \mu(p_H) = [\![H]\!]^\rho = [\![p]\!]^\rho = \mu(p) = [\![p]\!]^\mu$.
  Finalement $[\![p_H]\!]^\mu = [\![p]\!]^\mu$ donc $[\![R^0_H]\!]^\mu = \mathsf{V}$.
- Si $H = \neg H_1$, alors $R^0_H = p_H \leftrightarrow \neg p_{H_1}$, or $[\![p_H]\!]^\mu = \mu(p_H) = [\![H]\!]^\rho = [\![\neg H_1]\!]^\rho = \overline{[\![H_1]\!]^\rho} = \overline{\mu(p_{H_1})} = \overline{[\![p_{H_1}]\!]^\mu} = [\![\neg p_{H_1}]\!]^\mu$.
  Finalement $[\![p_H]\!]^\mu = [\![\neg p_{H_1}]\!]^\mu$ donc $[\![R^0_H]\!]^\mu = \mathsf{V}$.
- Si $H = H_1 \wedge H_2$, alors $R^0_H = p_H \leftrightarrow (H_1 \wedge H_2)$, or $[\![p_H]\!]^\mu = \mu(p_H) = [\![H]\!]^\rho = [\![H_1 \wedge H_2]\!]^\rho = [\![H_1]\!]^\rho [\![H_2]\!]^\rho = \mu(p_{H_1})\mu(p_{H_2}) = [\![p_{H_1}]\!]^\mu [\![p_{H_2}]\!]^\mu = [\![p_{H_1} \wedge p_{H_2}]\!]^\mu$.
  Finalement $[\![p_H]\!]^\mu = [\![p_{H_1} \wedge p_{H_2}]\!]^\mu$ donc $[\![R^0_H]\!]^\mu = \mathsf{V}$.
- De même pour les autres cas.

Ceci étant vrai pour toute sous-formule H de G, on en conclut que $[\![R]\!]^\mu = [\![\bigwedge_{H\in\mathcal{S}} R_H]\!]^\mu = \mathsf{V}$. □

♣. On remarque que G_H est définie par disjonction de cas sur H, et non par induction.

Lemme 3.43

Soit $\mu \in \mathbb{B}^{\widetilde{\mathcal{Q}}}$. Notons $\rho = \mu_{|\mathcal{Q}}$.
Si μ satisfait R alors pour tout $H \in \mathcal{S}$, $[\![H]\!]^\rho = \mu(p_H)$.

Démonstration : Supposons que $[\![R]\!]^\mu = \mathsf{V}$. Remarquons que pour tout $H \in \mathcal{S}$, $[\![R_H]\!]^\mu = V$, donc $[\![p_H \leftrightarrow G_H]\!]^\mu = \mathsf{V}$ donc $[\![p_H]\!]^\mu = \mu(p_H) = [\![G_H]\!]^\mu$. Soit la propriété $\mathscr{P}_H : [\![H]\!]^\rho = \mu(p_H)$. Démontrons que pour tout $H \in \mathcal{S}$, $\mathscr{P}_H$ est vraie, par induction sur les sous-formules de G.

- Si $H = \top$ est dans $\mathcal{S}$ alors $G_H = \top$, or $\mu(p_H) = [\![G_H]\!]^\mu$ donc $\mu(p_H) = \mathsf{V} = [\![\top]\!]^\rho = [\![H]\!]^\rho$.
- Si $H = \bot$ est dans $\mathcal{S}$ alors $G_H = \bot$, or $\mu(p_H) = [\![G_H]\!]^\mu$ donc $\mu(p_H) = \mathsf{F} = [\![\bot]\!]^\rho = [\![H]\!]^\rho$.
- Si $H = p \in \mathcal{Q}$ est dans $\mathcal{S}$ alors $G_H = p$, or $\mu(p_H) = [\![G_H]\!]^\mu$ donc $\mu(p_H) = \mu(p) = \rho(p) = [\![p]\!]^\rho = [\![H]\!]^\rho$.
- Si $H = \neg H_1$ est dans $\mathcal{S}$ alors H_1 est dans $\mathcal{S}$ et on suppose $\mathscr{P}_{H_1}$ vrai, à savoir : $[\![H_1]\!]^\rho = \mu(p_{H_1})$. Alors $G_H = \neg p_{H_1}$, or $\mu(p_H) = [\![G_H]\!]^\mu$ donc $\mu(p_H) = [\![\neg p_{H_1}]\!]^\mu = \overline{[\![p_{H_1}]\!]^\mu} = \overline{\mu(p_{H_1})} = \overline{[\![H_1]\!]^\rho} = [\![\neg H_1]\!]^\rho = [\![H]\!]^\rho$.
- Si $H = H_1 \wedge H_2$ est dans $\mathcal{S}$ alors H_1 et H_2 sont dans $\mathcal{S}$ et on suppose $\mathscr{P}_{H_1}$ et $\mathscr{P}_{H_2}$ vrais, à savoir : $[\![H_1]\!]^\rho = \mu(p_{H_1})$ et $[\![H_2]\!]^\rho = \mu(p_{H_2})$. Alors $G_H = p_{H_1} \wedge p_{H_2}$, or $\mu(p_H) = [\![G_H]\!]^\mu$ donc $\mu(p_H) = [\![p_{H_1} \wedge p_{H_2}]\!]^\mu = [\![p_{H_1}]\!]^\mu [\![p_{H_2}]\!]^\mu = \mu(p_{H_1})\mu(p_{H_2}) = [\![H_1]\!]^\rho [\![H_2]\!]^\rho = [\![H_1 \wedge H_2]\!]^\rho = [\![H]\!]^\rho$.
- De même pour les autres cas.

On a donc bien établi, par induction, que toute sous-formule H de G est telle que : $[\![H]\!]^\rho = \mu(p_H)$. □

On considère finalement la formule $\tilde{G} = R \wedge p_G$. Cette formule est une 3-CNF. De plus $\tilde{G}$ est satisfiable si et seulement si G est satisfiable. En effet :

⇒ Si $\tilde{G}$ est satisfiable, soit μ tel que $[\![\tilde{G}]\!]^\mu = \mathsf{V}$, alors $[\![R]\!]^\mu = \mathsf{V}$ et $[\![p_G]\!]^\mu = \mu(p_G) = \mathsf{V}$. Soit alors $\rho = \mu_{|\mathcal{Q}}$. Du lemme 3.43 on déduit que pour tout $H \in \mathcal{S}$, $[\![H]\!]^\rho = \mu(p_H)$, en particulier $[\![G]\!]^\rho = \mu(p_G) = \mathsf{V}$. Donc G est satisfiable.

⇐ Si G est satisfiable, soit ρ tel que $[\![G]\!]^\rho = \mathsf{V}$, soit alors $\mu \in \tilde{\mathcal{Q}} \to \mathbb{B}$ tel que $\mu_{|\mathcal{Q}} = \rho$ et $[\![R]\!]^\mu = \mathsf{V}$ (cette existence découle du lemme 3.42). En appliquant alors le lemme 3.43 à μ on en déduit que pour tout $H \in \mathcal{S}$, $[\![H]\!]^\rho = \mu(p_H)$, en particulier $[\![G]\!]^\rho = \mu(p_G) = \mathsf{V}$. Finalement $[\![\tilde{G}]\!]^\mu = [\![R \wedge p_G]\!]^\mu = [\![R]\!]^\mu [\![p_G]\!]^\mu = \mathsf{V} \cdot \mathsf{V} = \mathsf{V}$.

Une disjonction de 3 littéraux est de taille au plus $\underbrace{3}_{\text{variables}} + \underbrace{3}_{\neg} + \underbrace{2}_{\vee} = 8$. La 3-CNF $\tilde{G}$ que l'on vient de construire contient au plus $8|\mathcal{S}| + 1$ telles disjonctions. Finalement la formule $\tilde{G}$ est de taille au plus $\underbrace{8}_{\text{taille disjonction}} (8|\mathcal{S}| + 1) + \underbrace{8|\mathcal{S}|}_{\text{nombre de connecteurs } \wedge}$.

Ainsi il existe un algorithme de complexité polynomiale calculant la formule $\tilde{G}$ à partir de la formule G. Finalement le problème SAT se réduit en temps polynomial au problème 3SAT, qui est donc NP-difficile. □

Corollaire 3.44

Pour tout $n \geqslant 3$, le problème n-CNF-SAT est NP-difficile.

Démonstration : Puisque $n \geqslant 3$, une 3-CNF est en particulier une n-CNF, ainsi l'identité constitue une réduction de 3SAT à n-CNF-SAT. Sachant que 3SAT est NP-difficile, on en déduit que n-CNF-SAT l'est aussi. □

Définition 3.45

*Un problème Q est dit **NP-complet** s'il est dans NP et qu'il est NP dur.*

Exemple 3.46

SAT est un problème NP-complet. 3SAT est aussi un problème NP-complet : la propriété 3.39 montre qu'il est NP-difficile, et il est dans NP car il se réduit à SAT qui est dans NP.