Chapitre 4

Calculabilité, Décidabilité, Complexité

Hugo Salou MPI*

Dernière mise à jour le 13 mai 2023

Table des matières

Au chapitre 1, on s'est intéressé aux langages réguliers, et aux automates qui est un modèle de calcul relativement simple. Dans ce chapitre, on s'intéresse à un modèle plus puissant : un *ordinateur*, on va notamment re-définir la notion de *problème*. Le choix classique de définition d'un ordinateur n'est pas celui qui a été choisi au programme : un programme OCaml.

0 Remarques mathématiques

Définition : Étant donné une relation $\mathcal{R}$ sur $\mathcal{E} \times \mathcal{F}$, on dit que $\mathcal{R}$ est

— *totale à gauche* dès lors que $\forall e \in \mathcal{E},\ \exists f \in \mathcal{F},\ (e, f) \in \mathcal{R}$;

— *déterministe* dès lors que $\forall e \in \mathcal{E},\ \forall (f, f') \in \mathcal{F}^2$, si $(e, f) \in \mathcal{R}$ et $(e, f') \in \mathcal{R}$, alors $f = f'$.

Définition : On appelle *fonction totale* de $\mathcal{E}$ dans $\mathcal{F}$ une relation sur $\mathcal{E} \times \mathcal{F}$ déterministe et totale à gauche.

Définition : On appelle *fonction partielle* de $\mathcal{E}$ dans $\mathcal{F}$ une relation sur $\mathcal{E} \times \mathcal{F}$ déterministe. On note alors

$$\mathrm{def}(f) = \{x \in \mathcal{E} \mid \exists y \in \mathcal{F},\ (x, y) \in f\}.$$

Remarque :
Soit f une fonction partielle de $\mathcal{E}$ dans $\mathcal{F}$ tel que $\square \notin \mathcal{F}$, alors on peut completer f en une fonction totale de $\mathcal{E}$ dans $\mathcal{F} \cup \{\square\}$ de la manière suivante :

$$\begin{aligned} f : \mathcal{E} &\longrightarrow \mathcal{F} \cup \{\square\} \\ x &\longmapsto \begin{cases} f(x) & \text{si } x \in \mathrm{def}(f) \\ \square & \text{sinon.} \end{cases} \end{aligned}$$

1 Problèmes

Définition : Étant donnés un ensemble d'entrée $\mathcal{E}$, un ensemble de sortie $\mathcal{S}$, on appelle *problème* sur $\mathcal{E} \times \mathcal{S}$ une relation $\mathcal{R}$ [1] sur $\mathcal{E} \times \mathcal{S}$ totale à gauche.

Exemple :
Le problème

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

est donc défini comme

$$\textsc{Prime} = \{(0, \boldsymbol{F}), (1, \boldsymbol{F}), (2, \boldsymbol{V}), (3, \boldsymbol{V}), (4, \boldsymbol{F}), (5, \boldsymbol{V}), \ldots\} \subseteq \mathbb{N} \times \mathbb{B}.$$

Exemple :

1. C'est l'ensemble des liens entrées/sorties

Le problème

$$\textsc{Find}_0 : \begin{cases} \textbf{Entrée} & : \text{un tableau } T \text{ contenant au moins un } 0 \\ \textbf{Sortie} & : i \in \mathbb{N} \text{ tel que } T[i] = 0 \end{cases}$$

est donc défini comme

$$\textsc{Find}_0 = \Big\{ ([|0,1,1|], 0), ([|0,1,0|], 0), ([|0,1,0|], 2), \dots \Big\}.$$

Remarque :
De la même manière qu'en mathématiques, on n'écrit pas $\{(0,1),(1,2),(2,3),\dots\}$ mais $x \mapsto x+1$, en informatique, on préfère la notation sous forme de problème.

Remarque :
On précisera, en cas d'ambigüité la représentation choisie pour les entrées.

Exemple :
Les deux problèmes

$$\begin{cases} \textbf{Entrée} & : n \text{ sous forme décomposée en facteurs premiers} \\ \textbf{Sortie} & : n \text{ est-il premier ?} \end{cases}$$

et

$$\begin{cases} \textbf{Entrée} & : n \text{ en base } 2 \\ \textbf{Sortie} & : n \text{ est-il premier ?} \end{cases}$$

sont différents.

Remarque :
Lorsque ce n'est pas précisé, dans la suite du cours, les entiers sont représentés en base 2.

Exemple :
Dans le problème

$$\begin{cases} \textbf{Entrée} & : L_1 \text{ et } L_2 \text{ deux langages réguliers} \\ \textbf{Sortie} & : L_1 = L_2, \end{cases}$$

on doit préciser la représentation de L_1 et L_2.

Définition : On appelle *problème de décision* un problème à valeurs dans $\mathbb{B}$ déterministe.

Exemple :
Par exemple, le problème Prime est un problème de décision.

Remarque (Notation) :
Lorsque Q est un problème, on note $\mathscr{E}_Q$ son espace d'entrée, et $\mathscr{S}_Q$ son espace de sortie. De plus, si Q est un problème de décision, on note $Q^+ = \{e \in \mathscr{E}_Q \mid (e, \boldsymbol{V}) \in Q\}$ et $Q^- = \{e \in \mathscr{E}_Q \mid (e, \boldsymbol{F}) \in Q\}$. $\{Q^+, Q^-\}$ est une partition de $\mathscr{E}_Q$.

2 Décidabilité

La définition d'un algorithme comme une suite finie d'instruction élémentaire, nous montre que l'ensemble d'algorithmes est dénombrable. [2] Mais, l'ensemble de problèmes est indénombrable. En effet, pour $x \in [0, 1[$, on définit le problème

$$\textsc{Bit}_x : \begin{cases} \textbf{Entrée} & : n \in \mathbb{N} \\ \textbf{Sortie} & : \text{le } n\text{-ième bit de } x. \end{cases}$$

Et, comme $[0, 1[$ n'est pas dénombrable, l'ensemble des problèmes ne l'est pas non plus.

2.1 Modèles de calcul

Définition : On appellera *modèle de calcul* la donnée d'un ensemble de <u>machines</u>
- qu'il est possible d'exécuter sur des <u>entrées</u> ;
- qui peuvent <u>ou pas</u> retourner une réponse.

Exemple :
Les automates déterministes qu'il est possible d'exécution sur une entrée $w \in \Sigma^\star$, obtenant ainsi un booléen $b \in \mathbb{B}$, est un modèle de calcul.

Dans ce chapitre, notre modèle de calcul sera l'ensemble des fonction OCaml ayant pour type `string` $\to$ `string` qu'il est possible d'exécuter, qui peuvent donner un réponse ou non (boucle infinie, erreur).

Remarque :
On se place dans un monde d'exécution idéal : *mémoire infinie*.

Exemple :
On reprend le problème Prime. On code, sous forme de fonction OCaml, la machine ci-dessous. Elle répond au problème Prime.

```
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
    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)
```

Code 1 – Machine décidant le problème Prime

2. en bijection avec $\mathbb{N}$

Remarque (Notation) :
Dans la suite, lorsque $\mathscr{M}$ est une machine, et $w \in$ `string`, on notera

- $w \underset{\mathscr{M}}{\longrightarrow} w'$ si l'exécution de $\mathscr{M}$ sur w conduit à $w' \in$ `string`.
- $w \underset{\mathscr{M}}{\longrightarrow} \circlearrowleft$ si l'exécution de $\mathscr{M}$ sur w conduit à une erreur.

Dans la suite de ce chapitre, on fixe $\Sigma =$ `char` et donc $\Sigma^\star =$ `string`.

Remarque :
On pourra, dans la suite, généraliser la signature de nos machines à un type $\mathscr{E} \to \mathscr{F}$ dès lors qu'on exhibe une fonction de sérialisation $\varphi : \mathscr{E} \to \Sigma^\star$ inversible (sur son espace image) injective : avec $\varphi_{\mathscr{E}} : \mathscr{E} \to \Sigma^\star$ et $\varphi_{\mathscr{F}} : \mathscr{F} \to \Sigma^\star$, on a

```
let ma_super_fonction (e: 𝓔) : 𝓕 = ...

let ma_fonction (s: string) : string =
  φ_𝓕(ma_super_fonction(φ_𝓔^{-1}(s)))
```

Code 2 – Généralisation des machines ayant pour entrée un ensemble $\mathscr{E}$ et sortie $\mathscr{F}$

2.2 Décidabilité

Définition : Une fonction partielle $f : \mathscr{E} \to \mathscr{S}$ est dite *calculée* par une machine $\mathscr{M}$ dès lors que

$$\forall e \in \operatorname{def}(f), \quad e \underset{\mathscr{M}}{\longrightarrow} f(e).$$

On dit alors d'une telle fonction qu'elle est *calculable*.

Remarque :
Cette définition ne spécifie aucunement le comportement de $\mathscr{M}$ sur une entrée $e \notin \operatorname{def}(f)$.

Exemple :
Considérons la fonction partielle $\sqrt{\ } : \mathbb{Z} \to \mathbb{Z}$ telle que $\operatorname{def}(\sqrt{\ }) = \{p^2 \mid p \in \mathbb{N}\}$, et $\sqrt{x}$ est l'unique $y \in \mathbb{N}$ tel que $y^2 = x$.

```
let sqrt (n: int): int =
  if m < 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
```

Code 3 – Machine calculant la fonction $\sqrt{\ }$

— Pour n < 0, on a n $\underset{\mathscr{M}}{\longrightarrow} -1$.
— Pour n $\geqslant 0$ qui n'est pas un carré, n $\underset{\mathscr{M}}{\longrightarrow} \circlearrowleft$.
— Pour n $\geqslant 0$ qui est un carré, n $\underset{\mathscr{M}}{\longrightarrow} \sqrt{\mathtt{n}}$.

Ainsi, $\sqrt{\ }$ est calculable.

Définition : Étant donné qu'un problème de décision Q est un cas particulier de fonction totale $\mathscr{E}_Q \to \mathbb{B}$, on dit que Q est *décidé* par une machine $\mathscr{M}$ dès lors que

$$\forall e \in \mathscr{E}_Q, \qquad \left(e \in Q^+ \iff e \underset{\mathscr{M}}{\longrightarrow} \boldsymbol{V} \quad \text{et} \quad e \in Q^- \iff e \underset{\mathscr{M}}{\longrightarrow} \boldsymbol{F}\right).$$

On dit alors que ce problème Q est *décidable*.

Exemple :
On considère le problème

$$\textsc{Existe}_0 : \begin{cases} \textbf{Entrée} & : \text{un tableau } T \\ \textbf{Sortie} & : \exists i \in \mathbb{N},\ T[i] = 0 ?. \end{cases}$$

La machine ci-dessous décide le problème $\textsc{Existe}_0$.

```
exception OK

let existe (t: int array) : bool =
  try
    for i = 0 to (Array.length t) - 1 do
      if t.(i) = 0 then
        raise OK
    done;
    false
  with
  | OK -> true
```

Code 4 – Machine décide le problème $\textsc{Existe}_0$

2.3 Langages et problèmes de décision

Les notions de langages et problèmes de décision d'entrée $\Sigma^\star$ coïncident. En effet, à un langage $L \subseteq \Sigma^\star$, on associe le problème de décision

$$\textsc{Appartient}_L : \begin{cases} \textbf{Entrée} & : w \in \Sigma^\star \\ \textbf{Sortie} & : w \in L\ ?. \end{cases}$$

On a alors $(\textsc{Appartient}_L)^+ = L$. Réciproquement, à un problème de décision Q d'entrées $\Sigma^\star$, on associe le langage Q^+.

Définition : Un langage L est dit *décidable* lorsque le problème $\textsc{Appartient}_L$ est décidable.

Définition : Étant donné une machine $\mathscr{M}$ de type `string` $\to$ `bool`, on appelle *langage* de $\mathscr{M}$, que l'on note $\mathscr{L}(\mathscr{M})$, l'ensemble

$$\{w \in \Sigma^\star \mid w \underset{\mathscr{M}}{\longrightarrow} \boldsymbol{V}\}.$$

Remarque :
$\mathscr{L}(\mathscr{M})$ n'est pas le complémentaire de $\{w \in \Sigma^\star \mid w \underset{\mathscr{M}}{\longrightarrow} \boldsymbol{F}\}$. En effet, il peut exister $w \in \Sigma^\star$ tel que $w \underset{\mathscr{M}}{\longrightarrow} \circlearrowleft$.

Propriété : Un langage L est décidable si, et seulement si L est le langage d'une machine $\mathscr{M}$ telle que $\forall w \in \Sigma^\star$, $w \underset{\mathscr{M}}{\longrightarrow} \boldsymbol{V}$ ou $w \underset{\mathscr{M}}{\longrightarrow} \boldsymbol{F}$.

Propriété : Tout langage régulier est décidable.

Preuve :
Soit L un langage régulier. Montrons que L est décidable. On utilise alors la fonction OCaml suivante de reconnaissance d'un mot dans un automate (que l'on a déjà codé en TP). □

Propriété (stabilité des langages décidables) **:** Un langage décidable est stable par

1. union ;
2. intersection ;
3. complémentaire.

Preuve : 1. Soient L_1 et L_2 deux langages décidables. Montrons que $L_1 \cup L_2$ est décidable. Soit $\texttt{decide}_1$: string → bool la fonction décidant le langage L_1.[3] Soit $\texttt{decide}_2$: string → bool la fonction décidant du langage L_2. On construit alors la fonction

```
let decide (w : string) : bool =
  (decide₁ w) || (decide₂ w)
```

Code 5 – Fonction OCamlreconnaissant l'union de deux langages décidables

□

Remarque :
$\varnothing$ est décidable (par `fun s -> false`) ; $\Sigma^\star$ est décidable (par `fun s -> true`).

2.4 Sérialisation

Définition : Étant donné un type OCaml `t`, on appelle *sérialisation calculable* de ce type `t`, la donnée d'une fonction `f` OCaml de type `t` → `string` qui soit telle que

- pour tout e : `t`, (`f` e) est bien parenthésée ;
- `f` est injective ;

3. *i.e.* $\forall w \in \Sigma^\star$, $w \in L_1 \iff \texttt{decide}_1(w) = \texttt{true}$

— la réciproque de f (définie sur Im(f)) est définissable en OCaml.

Exemple :
Le type `int` est sérialisable par la fonction `string_of_int`.

Propriété : Soit t_a et t_b deux types OCaml sérialisables, alors le type $t_a \star t_b$ est sérialisable.

Preuve :
Soit φ_a et φ_b les fonctions de sérialisation des types t_a et t_b. On définit alors la fonction

```
let φ ((a,b): tₐ * t_b) =
  "(" ^ (φₐ a) ^ "),(" ^ (φ_b b) ^ ")"
```

Code 6 – Fonction OCaml sérialisant le produit cartésien de deux types sérialisables

On remarque que

— φ est à valeur dans les chaînes de caractères bien parenthésées ;
— φ est injective (par identification de parenthèses et injectivité de φ_a et φ_b) ;
— la réciproque de φ est décidable en OCaml (preuve à faire en OCaml).

□

Remarque :
Une programme OCaml est trivialement sérialisable : c'est déjà une chaîne de caractères.

Exemple :
La sérialisation de la fonction

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

Code 7 – Fonction factorielle en OCaml

est la chaîne de caractère

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

2.5 Machine universelle

Soit l'ensemble $\mathfrak{O}$ des chaînes de caractères qui sont des sérialisations de programme OCaml valide.

Exemple :
La sérialisation de la fonction `fact` trouvée précédemment est un élément de $\mathfrak{O}$. Mais, "`let`" $\notin \mathfrak{O}$.

Définition : Soit la fonction `interprète` : $\mathcal{O} \times \Sigma^\star \to \Sigma^\star$ définie par

$$\texttt{interprète}(\mathcal{M}, w) = \begin{cases} w' \text{ tel que } w \underset{\mathcal{M}}{\longrightarrow} w' \\ \text{non défini sinon.} \end{cases}$$

Théorème : La fonction `interprète` est calculable. On appelle *machine universelle* un programme OCaml la calculant.

Preuve :
On considère `utop` ou `WinCaml`. □

De même, considérons le problème suivant

$$\begin{cases} \textbf{Entrée} & : M \in \mathcal{O}, w \in \Sigma^\star, n \in \mathbb{N} \\ \textbf{Sortie} & : M \text{ se termine-t-elle sur } w \text{ en moins de } n \text{ étapes élémentaires ?} \end{cases}$$

Ce problème est décidable.

2.6 Théorème de l'Arrêt

Dans cette sous-section, on considère le problème

$$\text{Arrêt} : \begin{cases} \textbf{Entrée} & : M \in \mathcal{O}, w \in \Sigma^\star \\ \textbf{Sortie} & : M \text{ s'arrête-t-elle sur } w.\text{[4]} \end{cases}$$

Théorème : Le problème Arrêt est indécidable.

Preuve :
Par l'absurde : supposons Arrêt décidable. Soit `arret : string` $\to$ `bool` prenant en argument une chaîne de caractères qui est la sérialisation d'un couple M code d'une machine ($M \subseteq \Sigma^\star$), et $w \in \Sigma^\star$, et retournant `true` si et seulement si M s'arrête sur l'entrée w. Si l'entrée n'est pas une sérialisation convenable, on retourne `false`. On crée le programme suivant

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

Code 8 – Programme paradoxe prouvant que le problème Arrêt est indécidable

où `serialise_couple` est une fonction sérialisant un couple avec l'algorithme trouvé précédemment. Soit S_{paradoxe} la sérialisation de la fonction `paradoxe`.

Quid de (`paradoxe` S_{paradoxe}) : soit $c =$ (`serialise_couple` S_{paradoxe} S_{paradoxe}). La chaîne c est donc la sérialisation d'un couple dont la première composante est le code de la fonction `paradoxe`. De plus, `arret` se termine sur toute entrée.

- Si (`arret` c) $=$ `false`, alors on va dans la branche `else` et l'exécution de `paradoxe` sur S_{paradoxe} termine. Mais, comme (`arret` c) $=$ `false`, `paradoxe` ne se termine pas sur S_{paradoxe}.
- Si (`arret` c) $=$ `true`, alors on va dans la branche `then`, et donc (`paradoxe` S_{paradoxe}) ne se termine pas. Mais, comme (`arret` c) $=$ `true`, alors (`paradoxe` S_{paradoxe}) se termine.

On en conclut que le problème Arrêt est indécidable. □

4. *i.e.* $\exists ? w' \in \Sigma^\star,\ w \underset{M}{\longrightarrow} w'$

Corollaire : Il existe des problèmes indécidables.

2.7 Réduction

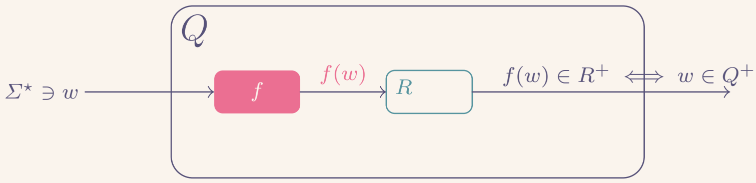

Figure 1 – Structure d'un sous-problème

Définition : Soit Q et R deux problèmes de décision. On dit que Q *se réduit au problème* R s'il existe $f : \mathscr{E}_Q \to \mathscr{E}_R$ totale et calculable, telle que

$$w \in Q^+ \iff f(w) \in R^+.$$

On note alors $Q \preccurlyeq R$.

Propriété : Si $Q \preccurlyeq R$, et que R est décidable, alors Q est décidable.

Preuve :
Soit $\texttt{decide}_R$: string $\to$ bool décidant R *i.e.* $(\texttt{decide}_R\ w) = \texttt{true} \iff w \in R^+$. Soit $f : \mathscr{E}_Q \to \mathscr{E}_R$ totale calculable, telle que $w \in Q^+ \iff f(w) \in R^+$. On doit coder la fonction suivante.

```
let decide_Q (w: string) : bool =
  (decide_R (f w))
```

Code 9 – Fonction décidant un sous-problème

La fonction $\texttt{decide}_Q$ décide bien le problème Q. En effet,

$$\begin{aligned}(\texttt{decide}_Q\ \texttt{w}) = \texttt{true} &\iff (\texttt{decide}_R\ (f\ \texttt{w}))\\ &\iff (f\ \texttt{w}) \in R^+\\ &\iff w \in Q^+.\end{aligned}$$

□

Corollaire : Si $Q \preccurlyeq R$, et Q non décidable, alors R non décidable.

Exemple :
On considère le problème

$$\textsc{NonVide} : \begin{cases}\textbf{Entrée} & : M \in \mathscr{O}\\ \textbf{Sortie} & : \mathscr{L}(M) \neq \varnothing\ ?\end{cases}.$$

Le problème NonVide est indécidable.

Exemple :
Les problèmes

$$\begin{cases}\textbf{Entrée} & : M_1 \text{ et } M_2 \text{ deux machines}\\ \textbf{Sortie} & : \mathscr{L}(M_1) \cup \mathscr{L}(M_2) = \Sigma^\star\end{cases}$$

et

$$\begin{cases}\textbf{Entrée} & : M_1 \text{ et } M_2 \text{ deux machines}\\ \textbf{Sortie} & : \mathscr{L}(M_1) \cap \mathscr{L}(M_2) = \varnothing\end{cases}$$

sont indécidables.

Propriété : La relation $\preccurlyeq$ est un *pré-ordre* :
- $\preccurlyeq$ est réflective ;
- $\preccurlyeq$ est transitive.

Preuve :
Soit Q un problème de décision.
- $Q \preccurlyeq Q$ par la fonction identité, qui est totale et calculable.
- Soient Q, R et S trois problèmes de décision tels que $Q \preccurlyeq R$ et $R \preccurlyeq S$. Soit donc f_1 la réduction de Q à R, et f_2 la réduction de R à S. Soit $f = f_2 \circ f_1 : \mathscr{E}_Q \to \mathscr{E}_S$. La fonction f est totale comme composée de fonctions totales, f est calculable comme composée de fonctions calculables. De plus,

$$\begin{aligned}\forall e \in \mathscr{E}_Q, \qquad f(e) \in S^+ &\iff f_2(f_1(e)) \in S^+\\ &\iff f_1(e) \in R^+\\ &\iff e \in Q^+\end{aligned}$$

□

3 Classe P et NP

Pour répondre à un problème, on peut le résoudre par des algorithmes plus ou moins rapides. Mais, l'objectif de cette section est de montrer que certains problèmes ne peuvent se résoudre que par des algorithmes lents, et que l'on ne peut pas faire mieux.

Définition : Le modèle de calcul impose une représentation des entrées par chaînes de caractères. Cela induit donc une notion de *taille d'entrée*, qui est la longueur de la chaîne de caractères.

3.1 Complexité d'une machine

Définition : Étant donné une machine $\mathscr{M}$ et une entrée $w \in \Sigma^\star$, on note $C^{\mathscr{M}}(w)$ le nombre d'opérations élémentaires effectuées lors de l'appel de $\mathscr{M}$ sur w. Lorsque $w \underset{\mathscr{M}}{\longrightarrow} \circlearrowleft$, on définit $C^{\mathscr{M}} = +\infty$.

Pour $n \in \mathbb{N}$, on définit alors

$$C_n^{\mathscr{M}} = \max\{C^{\mathscr{M}}(w) \mid w \in \Sigma^n\}.$$

Remarque :
On a, $\forall n \in \mathbb{N}$, $C_n^{\mathscr{M}} \in \bar{\mathbb{N}} = \mathbb{N} \cup \{+\infty\}$.

Définition : Soit $f : \mathbb{N} \to \mathbb{N}$ une fonction totale et calculable. On note $\textsc{Time}(f)$ l'ensemble des machines $\mathscr{M}$ telles que

— $\mathscr{M}$ s'arrête sur toute entrée ;
— $\left(C_n^{\mathscr{M}}\right)_{n\in\mathbb{N}} = \mathcal{O}\left(\left(f(n)\right)_{n\in\mathbb{N}}\right)$.

3.2 Classe P

Définition : On dit d'une machine $\mathscr{M}$ qu'elle est de *complexité polynômiale* dès lors qu'il existe $k \in \mathbb{N}$ tel que $\mathscr{M} \in \textsc{Time}(n^k)$.

Définition : On dit d'une fonction (partielle ou non), qu'elle est *calculable en temps polynômial* dès lors qu'il existe une machine $\mathscr{M}$ de complexité polynômiale la calculant.

Exemple :
— l'identité ($n \mapsto n$)
— la fonction successeur ($n \mapsto n+1$)

Propriété : La composée de deux fonctions totales calculables en temps polynômial est une fonction totale calculable en temps polynômial.

Preuve :
Soient f_1 et f_2 deux telles fonctions. Soit donc `calcule₁ : string → string` et `calcule₂ : string → string` calculant respectivement f_1 et f_2. On pose la fonction ci-dessous

```
let calcul s = calcule₁ (calcule₂ s)
```

Code 10 – Machine calculant la composée en temps polynômial

dont le nombre d'opérations élémentaires est

$$C(\mathtt{s}) = \underbrace{C^2(\mathtt{s})}_{\text{calcul de } f_2(\mathtt{s})} + \overbrace{C^1(f_2(\mathtt{s}))}^{\text{calcul de } f_1(\mathtt{s})} + \underbrace{1}_{\text{composée}}.$$

Or, la fabrication d'une chaîne de caractères de taille n nécessite au moins n opérations élémentaires. Soit $p_1 \in \mathbb{N}$ et $p_2 \in \mathbb{N}$ tels que la complexité de `calcule₁` est $\mathcal{O}(n^{p_1})$ et la complexité de `calcule₂` est $\mathcal{O}(n^{p_2})$. On a donc, pour tout $w \in \Sigma^\star$ avec $|w| = n$, que $|f_2(w)| = \mathcal{O}(n^{p_2})$ Par composition des $\mathcal{O}$, on a que

$$C(\mathtt{s}) = \mathcal{O}(|\mathtt{s}|^{p_1 \cdot p_2}).$$

□

Remarque :
Dans la preuve précédente, l'ordre du polynôme change. En effet, la composée de deux programmes en $\mathcal{O}(n^2)$ est un $\mathcal{O}(n^4)$. L'espace des fonctions calculables en $\mathcal{O}(n^p)$, pour un p fixé, n'est pas stable par composition.

Définition (Classe P) : On dit qu'un problème est dans **P** dès lors qu'il est décidable en temps polynômial.

Propriété (Stabilité de la classe **P**) **:** La classe **P** est stable par
— union ; — intersection ; — complémentaire.

Preuve :
à faire □

3.3 Classe NP

Définition (Classe **NP**) **:** On dit qu'un problème de décision Q est dans **NP** si et seulement si

— il existe un polynôme A ;
— un problème Verif $\in \mathbf{P}$;
— un ensemble $\mathscr{C}$ (certificats),

tels que

$$\forall w \in \Sigma^\star, \quad \Big(w \in Q^+ \iff \exists u \in \mathscr{C},\ |u| \leqslant A(|w|) \text{ et } (w, u) \in \textsc{Verif}^+\Big).$$

La classe **NP** est la classe de problèmes tels qu'ils sont vérifiables en temps polynômial. Par exemple, si on a une formule logique, trouver un environnement propositionnel est coûteux en temps, MAIS vérifier la solution est très simple. Nous verrons ce résultat plus tard dans cette sous-section.

Le "**NP**" ne vient pas de **N**on **P**olynômial, mais vient de **N**on-déterministe **P**olynômial.

Propriété :

$$\mathbf{P} \subseteq \mathbf{NP}.$$

Preuve :
À faire : Modifier la définition de Verif avec la nouvelle définition d'un problème **NP** Soit Q un problème de décision dans **P**. On pose $\mathscr{C} = \mathscr{E}_Q$, $A(X) = X$, et Verif $= Q$. En effet, pour tout $w \in \Sigma^\star$,

$$w \in Q^+ \iff \exists u = w,\ |u| \leqslant A(|w|) \text{ et } u \in \textsc{Verif}^+.$$

□

Propriété :

$$\textsc{Sat} \in \mathbf{NP}.$$

Rappel **:**
On rappelle la définition du problème Sat.

$$\textsc{Sat} : \begin{cases} \textbf{Entrée} & : \text{Une formule } G \\ \textbf{Sortie} & : \text{Existe-t-il } \rho \in \mathbb{B}^{\mathrm{vars}(G)} \text{ tel que } [\![G]\!]^\rho = V\,? \end{cases}$$

Preuve :
Soit alors le problème suivant.

$$\textsc{VerifSat} : \begin{cases} \textbf{Entrée} & : \left(\begin{array}{l} \text{Une formule } G, \\ \text{un environnement propositionnel } \rho \in \mathbb{B}^{\mathrm{vars}(G)}, \end{array} \right. \\ \textbf{Sortie} & : [\![G]\!]^\rho \stackrel{?}{=} V \end{cases}$$

En TP, on a codé une solution polynômial à VERIFSAT donc VERIFSAT $\in$ **P**. On définit l'ensemble $\mathscr{C}$ des certificats comme

$$\mathscr{C} = \{(G, \rho) \mid G \in \mathscr{F} \text{ et } \rho \in \mathbb{B}^{\text{vars}(G)}\}.$$

On a alors

$$G \in \text{SAT}^+ \iff \exists \rho \in \mathbb{B}^{\text{vars}(G)},\ [\![G]\!]^\rho = \boldsymbol{V}.$$

Il suffit alors de choisir $A(X) = 2X$. □

3.4 NP-difficile

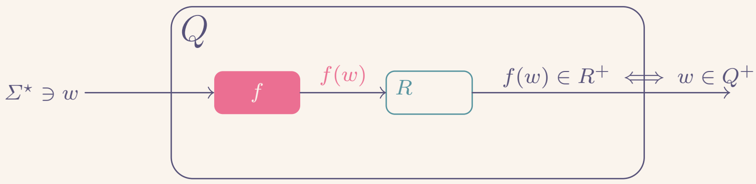

FIGURE 2 – Structure d'un sous-problème

Définition (Réduction polynômiale) **:** Soit Q et R deux problèmes de décision, on appelle *réduction polynômiale* de Q à R la donnée d'une fonction f totale, calculable en temps polynômial de $\mathscr{E}_Q$ dans $\mathscr{E}_R$ telle que

$$\forall w \in \mathscr{E}_Q, \quad w \in Q^+ \iff f(w) \in R^+.$$

On note alors $Q \preccurlyeq_{\mathrm{p}} R$.

Propriété : La relation $\preccurlyeq_{\mathrm{p}}$ est transitive et reflective : c'est un pré-ordre.

Preuve :
La réflectivité est assurée car id est totale et calculable en temps polynômial. La transitivité est assurée par les propriétés précédentes (composée de deux fonctions polynômiales ?). □

Propriété : Si $R \preccurlyeq_{\mathrm{p}} Q$, et $R \in \mathbf{P}$, alors $Q \in \mathbf{P}$. □

Définition (**NP**-difficile) **:** Un problème Q est **NP**-difficile si

$$\forall R \in \mathbf{NP},\ R \preccurlyeq_{\mathrm{p}} Q.$$

Propriété : Si Q est **NP**-difficile, et $Q \preccurlyeq_{\mathrm{p}} R$, alors R est **NP**-difficile.

Preuve :
Soit $S \in \mathbf{NP}$, donc $S \preccurlyeq_{\mathrm{p}} Q$. De plus, $Q \preccurlyeq_{\mathrm{p}} R$, donc $S \preccurlyeq_{\mathrm{p}} R$, par transitivité. Ceci étant vrai pour tout $S \in \mathbf{NP}$, on en déduit que R est **NP**-difficile. □

On admet le théorème suivant.

Théorème (Cook-Levin) : Le problème Sat est **NP**-difficile.

Preuve :
Admis □

Définition (n-fnc) : Soit $n \in \mathbb{N}$. Une formule G est sous forme n-fnc dès lors que G est sous forme fnc et chaque clause de G contient au plus n littéraux.

On parle aussi de forme cnf traduction anglaise de fnc. De même, on parle de forme n-cnf au lieu de n-fnc.

Exemple :
La formule $(p \lor q) \land r$ est une 2-cnf. La formule $(p \lor q \lor p) \land (r \lor p \lor q \lor q)$ est une 4-cnf.

Définition : On définit le problème ci-dessous.

$$n\text{-cnf-Sat} : \begin{cases} \textbf{Entrée} & : G \text{ une } n\text{-cnf} \\ \textbf{Sortie} & : \text{Existe-t-il } \rho \text{ tel que } [\![G]\!]^\rho = \boldsymbol{V}\,? \end{cases}$$

Propriété : Soit 3sat = 3-cnf-Sat. Le problème 3sat est **NP**-difficile.

Preuve (par réduction de Sat à 3sat) *:*
Soit G une formule sur $\mathbb{Q}$, un ensemble de variables propositionnelles. Pour toute sous-formule H, on note

- x_H une variable propositionnelle,
- K_H une formule définie par
  - si $H = \top$, $H = \bot$ ou $H = p \in \mathbb{Q}$, alors $K_H = H$,
  - si $H = \neg H_1$, alors $K_H = \neg x_{H_1}$,
  - si $H = H_1 \odot H_2$, avec $\odot \in \{\rightarrow, \lor, \land, \leftrightarrow\}$, alors $K_H = x_{H_1} \odot x_{H_2}$.

Définissons alors la formule

$$K = \bigwedge_{H \text{ sous-formule de } G} (x_H \leftrightarrow K_H).$$

On note aussi, si $\mathbb{Q} \subseteq \mathbb{Q}'$ deux ensembles de variables propositionnelles, et $\rho \in \mathbb{B}^{\mathbb{Q}}$, on note $\rho' \sqsupseteq \rho$ dès lors que $\operatorname{def}(\rho') = \mathbb{Q}'$ et $\forall x \in \mathbb{Q}$, $\rho(x) = \rho'(x)$. On pose $\mathbb{Q}' = \mathbb{Q} \cup \{x_H \mid H \text{ sous-formule de } G\}$. On considère à présent le lemme suivant.

Lemme : Soit $\rho \in \mathbb{B}^{\mathbb{Q}}$. Il existe $\rho' \in \mathbb{B}^{\mathbb{Q}'}$ tel que $\rho' \sqsupseteq \rho$ et $[\![K]\!]^{\rho} = \boldsymbol{V}$.

Prouvons ce lemme.

Preuve :
On définit

$$\rho'(x) = \begin{cases} \rho(x) & \text{si } x \in \mathbb{Q} \\ [\![H]\!]^{\rho} & \text{si } x = x_H \,. \end{cases}$$

Soit alors H une sous-formule de G.

— Si $H = \top$, alors $\rho'(x_H) = [\![H]\!]^{\rho} = [\![x_H]\!]^{\rho'}$, et $[\![K_H]\!]^{\rho'} = [\![H]\!]^{\rho'} = [\![\top]\!]^{\rho'} = [\![H]\!]^{\rho}$. Ainsi, $[\![x_H \leftrightarrow K_H]\!]^{\rho'} = \boldsymbol{V}$.

— Si $H = \neg H_1$, alors $[\![x_H]\!]^{\rho'} = [\![H]\!]^{\rho}$, et

$$[\![K_H]\!]^{\rho'} = [\![\neg x_{H_1}]\!]^{\rho'} = \overline{[\![x_{H_1}]\!]^{\rho'}} = \overline{[\![H_1]\!]^{\rho}} = [\![H]\!]^{\rho} \,.$$

On en déduit que $[\![x_H \leftrightarrow K_H]\!]^{\rho'} = \boldsymbol{V}$.

— Si $H = H_1 \wedge H_2$, alors $[\![x_H]\!]^{\rho'} = [\![H]\!]^{\rho}$, et

$$\begin{aligned} [\![K_H]\!]^{\rho'} &= [\![x_{H_1} \wedge x_{H_2}]\!]^{\rho'} \\ &= [\![x_{H_1}]\!]^{\rho'} \cdot [\![x_{H_2}]\!]^{\rho'} \\ &= [\![H_1]\!]^{\rho} \cdot [\![H_2]\!]^{\rho} \\ &= [\![H_1 \wedge H_2]\!]^{\rho} \\ &= [\![H]\!]^{\rho} \end{aligned}$$

— De même pour les autres cas. . .

On a donc

$$\begin{aligned} \Bigg[\!\!\Bigg[\bigwedge_{H \text{ sous-formule de } G} x_H \leftrightarrow K_H \Bigg]\!\!\Bigg]^{\rho'} &= \prod_{H \text{ sous-formule de } G} [\![x_H \leftrightarrow K_H]\!]^{\rho'} \\ &= \boldsymbol{V} \end{aligned}$$

et donc $[\![K]\!]^{\rho'} = \boldsymbol{V}$. □

Lemme : Pour tout environnement propositionnel $\rho \in \mathbb{B}^{\mathbb{Q}}$, et pour tout $\rho' \in \mathbb{B}^{\mathbb{Q}'}$. Si $[\![K]\!]^{\rho'} = \boldsymbol{V}$, alors $\rho(x_G) = [\![G]\!]^{\rho}$.

Preuve :
Soient $\rho \in \mathbb{B}^{\mathcal{Q}}$ et $\rho' \in \mathbb{B}^{\mathcal{Q}'}$ deux environnements propositionnels. *Montrons, par induction sur les sous-formules de G, pour toute sous-formule H de G, que* $\rho'(x_H) = [\![H]\!]^\rho$.

- Si $H = \top$, alors, comme $[\![K]\!]^{\rho'} = \boldsymbol{V}$, on a $[\![x_H \leftrightarrow K_H]\!]^{\rho'} = \boldsymbol{V}$, donc $\rho'(x_H) = [\![H]\!]^{\rho'} = [\![H]\!]^\rho$.
- Si $H = \neg H_1$, avec $\rho'(x_{H_1}) = [\![H_1]\!]^\rho$ par hypothèse d'induction, alors $[\![K]\!]^{\rho'} = \boldsymbol{V}$ donc $[\![x_H \leftrightarrow K_H]\!]^{\rho'} = \boldsymbol{V}$, d'où

$$\begin{aligned}\rho'(x_H) &= [\![K_H]\!]^{\rho'}\\ &= [\![\neg x_{H_1}]\!]^{\rho'}\\ &= \overline{[\![x_{H_1}]\!]^{\rho'}}\\ &= \overline{[\![H_1]\!]^{\rho}}\\ &= [\![\neg H_1]\!]^{\rho}\\ &= [\![H]\!]^{\rho}\end{aligned}$$

- Si $H = H_1 \wedge H_2$, avec $\rho'(x_{H_1}) = [\![H_1]\!]^\rho$, et $\rho'(x_{H_2}) = [\![H_2]\!]^\rho$ par hypothèse d'induction, alors $[\![K]\!]^\rho = \boldsymbol{V}$ donc $[\![x_H \leftrightarrow K_H]\!]^{\rho'} = \boldsymbol{V}$ d'où

$$\begin{aligned}\rho'(x_H) &= [\![K_H]\!]^{\rho'}\\ &= [\![x_{H_1} \wedge x_{H_2}]\!]^{\rho'}\\ &= [\![x_{H_1}]\!]^{\rho'} \cdot [\![x_{H_2}]\!]^{\rho'}\\ &= [\![H_1]\!]^{\rho} \cdot [\![H_2]\!]^{\rho}\\ &= [\![H_1 \wedge H_2]\!]^{\rho}\\ &= [\![H]\!]^{\rho}\end{aligned}$$

- De même pour les autres cas...

□

À l'instance G du problème Sat, on associe donc la formule $K \wedge x_G$.

"$\Longrightarrow$" Soit $G \in$ Sat$^+$, soit alors ρ tel que $[\![G]\!]^\rho = \boldsymbol{V}$, alors il existe, d'après le premier lemme, un environnement ρ' tel que $[\![K]\!]^{\rho'} = \boldsymbol{V}$. Le second lemme nous donne alors $[\![x_G]\!]^{\rho'} = [\![G]\!]^\rho = \boldsymbol{V}$ donc $[\![K \wedge x_G]\!]^{\rho'}$ d'où $[\![K \wedge x_G]\!] \in$ Sat$^+$.

"$\Longleftarrow$" Si $K \wedge x_G \in$ Sat$^+$, il existe donc ρ' tel que $[\![K \wedge x_G]\!]^{\rho'} = \boldsymbol{V}$ donc $[\![K]\!]^{\rho'} = \boldsymbol{V}$ et $\rho'(x_G) = \boldsymbol{V}$. Soit alors ρ la restriction de ρ' à $\mathcal{Q}$. Ainsi, d'après le second lemme, $[\![G]\!]^\rho = \rho'(x_G) = \boldsymbol{V}$.

□

Remarque :
Pour tout $n \geqslant 3$, le problème n-cnf-sat est **NP**-difficile. Ceci peut être prouvé par réduction avec la fonction identité à 3sat.

Définition : On dit qu'un problème est **NP**-complet s'il est dans la classe **NP** et dans la classe **NP**-difficile :

$$\textbf{NP}\text{-complet} = \textbf{NP}\text{-difficile} \cap \textbf{NP}.$$

Remarque :
Le problème Sat est **NP**-complet.