Calculabilité

Support de cours et de TD

Eugene Asarin

Grenoble – 2003

Table des matières

Préface

Ce document contient les notes de cours et les exercices de TDs de la Calculabilité, que j'ai faits en 2000-2003 pour la maîtrise d'informatique à l'Université Joseph Fourier.

En préparant mon cours de la Calculabilité je me suis basé sur le cours fait à l'UJF par Christian Boitet jusqu'à 1999, et j'ai profité de ses conseils.

Ce document n'aurait jamais vu le jour sans l'aide de deux jeunes chercheurs, Gerardo Schneider, qui étant chargé d'un groupe de TD en 2001, a assisté à tous les cours et a pris les notes très détaillées, et Thao Dang, qui a saisi ces notes en LaTeX.

Je remercie également les chargés de TDs, qui ont beaucoup influencé le contenu et la forme de mes cours. Ce sont (dans l'ordre chronologique) Peter Niebert, Gerardo Schneider, Christian Boitet et Yassine Lakhnech.

Ce document dans sa forme actuelle est encore incomplet et certainement contient quelques imprécisions. J'espère de le compléter et de l'améliorer dans les mois qui viennent. Je serai reconnaissant aux lecteurs pour leurs commentaires, remarques et suggestions constructives qu'ils peuvent m'envoyer à l'adresse `asarin@imag.fr`

Eugene Asarin
Grenoble, avril 2003

Remarque

Actuellement je n'enseigne pas le cours de la Calculabilité, mais je laisse ce document sur ma page web pour les étudiants et les enseignants qui peuvent le trouver utile. Je viens de corriger quelques petites fautes, merci en particulier à Ouri Maler qui m'a signalé une imprécision concernant les formes normales. Je serai toujours reconnaissant aux lecteurs pour leurs commentaires, qu'ils peuvent m'envoyer à l'adresse `asarin@liafa.jussieu.fr`

Eugene Asarin
Paris, avril 2006

Notations typographiques

Le texte sur le fond gris représente des raisonnements informels

Les exercices sont données en utilisant cette police

Introduction

0.1 Quelques exemples

Les cours d'informatique que vous avez eu peuvent donner une impression que pour chaque problème on peut trouver un algorithme de solution. Dans le cours de la Calculabilité nous montrerons que ce n'est pas le cas et que pour des nombreux problèmes naturels et intéressants il n'existe pas d'algorithme. Nous commencerons par quelques exemples illustratifs des problèmes de décision. Pour le moment nous ne donnons aucune preuve.

Problème de terminaison du programme "$3x + 1$" Est-ce que le programme ci-dessous s'arrête sur un x donné ?

while $x > 1$ **do** {
  if ($x \mod 2 = 0$) **then** $x := x/2$;
  else $x := 3x + 1$;
}

Réponse. Pour ce problème, on peut utiliser la procédure suivante : à partir de x donné exécuter ce programme, et s'il s'arrête retourner "OUI". Par contre, si pour un certain x le programme "$3x + 1$" boucle, cette procédure ne pourra jamais donner la réponse "NON". Une telle procédure s'appelle un *semi-algorithme*.

Il est inconnu s'il existe un algorithme de décision pour ce problème, qui pour chaque x donné renvoie une réponse correcte "OUI" ou "NON". Une conjecture dit que ce programme s'arrête toujours. Si cette conjecture est vraie l'algorithme de décision serait tout simplement de retourner tout de suite "OUI" pour chaque x.

Problème de terminaison de programme C'est une généralisation du problème précédent. Étant donné un texte d'une fonction (avec un argument entier) programmé en Pascal et un $x \in \mathbb{N}$, est-ce que cette fonction s'arrête pour l'argument x ?

Pour ce problème il existe un semi-algorithme (trouvez-le), mais il n'existe pas d'algorithme.

Problème de correction de programme Étant donné un texte d'une fonction (avec un argument entier) programmé en Pascal, est-ce que cette fonction calcule (par exemple) la factorielle.

Pour ce problème il n'existe ni un algorithme, ni même un semi-algorithme. Donc le problème (très pratique) si un programme donnée satisfait une spécification n'admet pas d'algorithme de décision.

Problème de logique 1. Est-ce qu'une formule propositionnelle $\mathcal{F}$ donnée (telle que $\mathcal{F} = p \Rightarrow g \vee p \vee \neg p$) est valide ?

Réponse. Il existe des algorithmes pour ce problème, par exemple,

- Algorithme 1 : Construire la table de vérité. Si toutes les lignes contiennent seulement '1', répondre "OUI". Si dans une ligne il y a '0' répondre "NON".
- Algorithme 2 : DP (Davis-Putnam), vu en cours de Logique.

Problème de logique 2. Est-ce qu'une formule du premier ordre donnée (telle que par exemple $\forall x P(x) \rightarrow \exists y P(y)$) est valide ?
Réponse. Il n'existe pas d'algorithme général (c-à-d il n'y a pas d'algorithme qui donne la bonne réponse pour une formule du premier ordre quelconque). La procédure vue en cours de Logique (à savoir, skolémiser la négation de la formule, la mettre en forme clausale et appliquer la résolution) peut ne pas terminer.

Problème de langages 1. Pour une grammaire hors-contexte Γ, est-ce que son langage est vide, c-à-d $L(\Gamma) = \emptyset$?
Réponse. Il existe des algorithmes (vus en cours d'Automates et Langages).

Problème de langages 2. Pour une grammaire hors-contexte Γ et un mot w, est-ce que $w \in L(\Gamma)$?
Réponse. Il existe des algorithmes (vus en cours d'Automates et Langages).

Problème de langages 3. Pour deux grammaires hors-contexte Γ_1 et Γ_2, est-ce que $L(\Gamma_1) = L(\Gamma_2)$?
Réponse. Il n'existe pas d'algorithme.

Problème de langages 4. Pour deux mots α et β et les règles de réécriture R (de la forme $v \rightarrow w$), est-il possible de transformer α en β en appliquant plusieurs fois les règles R ?
Réponse. Il n'existe pas d'algorithme.

0.2 Terminologie

Dans ce cours, nous allons considérer des problèmes de décision qui peuvent être formulés comme suit. Étant donné un ensemble U de toutes les données possibles (un univers) et un ensemble B de toutes les données dites "bien" ($B \subseteq U$), pour chaque élément $x \in U$, répondre "OUI" si $x \in B$ et "NON" si $x \notin B$. Par exemple, pour le problème de logique 2

$$\begin{aligned} U &= \{\text{ toutes les formules propositionnelles }\} \\ B &= \{\text{ toutes les formules valides }\}. \end{aligned}$$

Par abus de notation nous écrirons ce problème comme "$x \in B$?", ou comme "$B(x)$?"

Voici une définition informelle de problèmes décidables, semi-décidables, indécidables.

Définition 1 *Pour un problème $P = (U, B)$ donné,*
- *Le problème P est* décidable $\Leftrightarrow$ *Il existe un* algorithme *pour P (c'est-à-dire une procédure qui s'arrête et répond "OUI" si l'entrée $x \in B$ et répond "NON" si l'entrée $x \notin B$).*
- *Le problème P est* indécidable $\Leftrightarrow$ *Il n'existe* pas d'algorithme *pour P.*
- *Le problème P est* semi-décidable $\Leftrightarrow$ *Il existe un* semi-algorithme *pour P (c'est-à-dire une procédure telle que si l'entrée $x \in B$ elle répond "OUI" ; si l'entrée $x \notin B$ dit "NON" ou ne donne pas de réponse).*

Il est clair qu'un problème décidable est aussi semi-décidable. ◇

Remarque. Pour montrer qu'un problème est *décidable*, il suffit de trouver *un algorithme* (un seul suffit et ceci peut se faire sans utiliser la théorie de la calculabilité). Par contre, pour montrer qu'un problème est *indécidable*, il faut considérer *tous les algorithmes possibles* et montrer qu'aucun d'eux ne résout le problème, ce qui est plus difficile et impossible sans théorie et sans notion rigoureuse d'algorithme.

0.2.1 Exercices - sans théorie

Pour chacun de problèmes suivants donner explicitement le U et le B. Essayer de trouver
- un algorithme de décision ;
- à défaut, un algorithme de semi-décision ("un semi-algorithme") ;
- à défaut, une opinion argumentée, si un tel algorithme ou semi-algorithme existe.

1. Étant donné un graphe G et un entier k, y a-t-il une clique[1] de taille k dans G ?
2. Étant donnée une équation quadratique $ax^2 + bx + c = 0$ aux coefficients entiers, est-ce qu'elle a une solution réelle ? une solution entière ?
3. Étant donnée une équation algébrique[2] à une ou plusieurs variables et aux coefficients entiers, est-ce qu'elle a une solution réelle ? une solution entière ?
   Réfléchissez d'abord à la façon de résoudre l'équation
   $x^{66} - 15x^{62} - 2002 = 0$ *ou bien* $x^7 - 17xy + y^5 - 9999 = 0$.
4. Étant donnée la valeur initiale de x, est-ce que le programme suivant s'arrête :
   while $x \bmod 5 \neq 0$ **do**
   if $x \bmod 5 = 4$
   then $x := x^2 + 1$
   else $x := x + 2$
5. C'est une légère généralisation du problème précédent. Étant données la valeur initiale de x et la valeur de y, est-ce que le programme suivant s'arrête :
   while $x \bmod y \neq 0$ **do**
   if $x \bmod y = y - 1$
   then $x := x^2 + 1$
   else $x := x + 2$
6. Même question que pour 4 et 5
   while $x \neq 1$ **do**
   if $x \bmod 2 = 0$
   then $x := x/2$
   else $x := 3x - 1$

0.3 Calculabilité vs Complexité

Nous représentons une brève comparaison entre ces deux disciplines sous la forme d'un tableau

	Calculabilité	Complexité
Question :	*Existe*-t-il un algorithme ?	Existe-t-il un algorithme *rapide* ?
Hypothèse :	Ressources non-bornées	Ressources bornées
Technique 1 :	Diagonalisation	Diagonalisation
Technique 2 :	Réduction calculable	Réduction polynômiale

Le plan du cours

1. Définitions (rigoureuses) de la notion d'algorithme (modèles de calcul).
2. Propriétés générales des problèmes (décidabilité, indécidabilité, méthode de preuve de l'indécidabilité, etc.)
3. Applications

[1] un sous-ensemble de sommets de G dont chaque deux éléments sont reliés par une arête
[2] une équation de la forme **un polynôme**$= 0$

Chapitre 1

Définitions formelles de la notion d'algorithme et modèles de calcul

Notre but est de donner une définition formelle de la classe de fonctions calculables. Cette définition doit être à la fois intuitive et correspondre à la pratique de programmation.

Une définition simple est la suivante : une fonction f est calculable s'il existe un algorithme pour la calculer. Puis, on peut envisager de définir un algorithme par un programme PASCAL ou par un programme CAML.

L'inconvénient de cette définition est que la sémantique de ces langages est très compliquée (et peut varier d'une réalisation à une autre) et, en plus, ces langages sont trop riches pour des analyses théoriques. Pour cette raison nous définirons les modèles de calculs et de "langages de programmations" beaucoup plus simples, mais assez puissant pour décrire n'importe quel algorithme.

L'approche, que nous allons utiliser dans ce cours, est dite *fonctionnelle*. Nous allons restreindre notre attention aux algorithmes pour calculer les fonctions de la forme $f : \mathbb{N}^k \to \mathbb{N}$, et nous voulons aboutir à une définition (formelle) des *fonctions calculables* $f : \mathbb{N}^k \to \mathbb{N}$.

Pour voir le lien entre la décidabilité et la calculabilité, considérons un problème de décision (U, B) où $U = \mathbb{N}^k$. Le problème est donc le suivant : pour $x \in U$ décider si $x \in B$. Afin de décider si une entrée $x \in B$, nous allons utiliser la *fonction caractéristique* de l'ensemble B (d'entrées "bien") $\chi_B(x) : U \to \mathbb{N}$, qui est définie comme suit.

Définition 2 (Fonction caractéristique)

$$\chi_B(x) = \begin{cases} 1 & \textit{si } x \in B, \\ 0 & \textit{sinon.} \end{cases}$$

Définition 3 *Le problème (U, B) est décidable si et seulement si χ_B est* calculable.

Une fois que nous aurons une notion précise de fonction calculable, la définition précédente deviendra aussi rigoureuse.

1.1 Fonctions récursives primitives

Dans cette section nous faisons la première tentative de définir la classe des fonctions calculables. On obtiendra une classe assez large, contenant presque tous les algorithmes que vous connaissez, mais, néanmoins insuffisante. Dans les sections suivantes nous l'élargirons encore.

1.1.1 Définition de fonctions récursives primitives

Définition 4 (RP) *La classe des fonction récursives primitives (RP) est la classe de fonctions $f : \mathbb{N}^k \to \mathbb{N}$ définie inductivement en utilisant* 3 *fonctions de base et* 2 *constructeurs suivants.*

- *Fonctions de base*
  - Zéro $O : \mathbb{N}^0 \to \mathbb{N}$ *telle que* $O() = 0$.
  - Successeur $\sigma : \mathbb{N} \to \mathbb{N}$ *telle que* $\sigma(x) = x + 1$.
  - Projecteur $\pi_i^k : \mathbb{N}^k \to \mathbb{N}$ *telle que* $\pi_i^k(x_1, x_2, \ldots, x_k) = x_i$.
- *Constructeurs*
  - Composition $Comp(f, g_1, \ldots, g_k) = h$ *t.q.* $h(\bar{x}) = f(g_1(\bar{x}, \ldots, g_k(\bar{(x)}))$.
  - $Rec\_Pri(f, g) = u$ *t.q.*

$$\begin{cases} u(\bar{x}, 0) = f(\bar{x}) \\ u(\bar{x}, n+1) = g(\bar{x}, n, u(\bar{x}, n)) \end{cases}$$

Autrement dit, la classe RP est la plus petite classe qui contient les fonctions de base O, σ, π_i^k et est fermée par les constructeurs $Comp$ et Rec_Pri.

Exemple 1. Nous allons définir la fonction $Plus(x, y) = x + y$. D'abord,

$$\begin{cases} Plus(x, 0) = x \\ Plus(x, n+1) = Plus(x, n) + 1 \end{cases}$$

Il suffit maintenant de récrire les équations ci-dessus sous forme $Rec\_Pri(f, g)$. Nous avons

$$\begin{cases} Plus(x, 0) = \pi_1^1(x) = f(x) \\ Plus(x, n+1) = \sigma(\pi_3^3(x, n, Plus(x, n))) = g(x, n, Plus(x, n)) \end{cases}$$

Donc, $Plus = Rec\_Pri(\pi_1^1, Comp(\sigma, \pi_3^3))$. ◇

Exemple 2. $zero : \lambda x.0$ (avec un argument).

$$\begin{cases} zero(0) = 0 = O() = f() \\ zero(n+1) = zero(n) = \pi_2^2(n, zero(n)) = g(n, zero(n)) \end{cases}$$

Donc, $zero = Rec\_Pri(O, \pi_2^2)$. ◇

Dans les deux séries d'exercices 1.1.2 et 1.2.1 nous établirons la récursivité primitive de nombreuses fonctions utiles.

1.1.2 Exercices – quelques fonctions RP

Montrer que les fonctions suivantes sont récursives primitives (RP) en les construisant à partir de fonctions de base $0, \sigma, \pi$ avec les constructeurs Comp et Rec_Pri :

1. 7 (d'arité 0, c-à-d $\lambda.0$ et d'arité 1, c-à-d $\lambda x.7$) ;
2. Mult=$\lambda xy.x \times y$;
3. $\uparrow = \lambda xy.x \uparrow y$ (cela signifie x^y) ;
4. $\lambda xy.x \uparrow\uparrow y$ (cela signifie $\underbrace{x \uparrow (x \uparrow \ldots \uparrow x)}_{y \text{ fois}}$) ;
5. $\lambda x.sg(x)$ (signe : 0 quand x est nul, 1 quand x est positif) ;
6. $\lambda xy.x \dot{-} y$ (différence tronquée ou monus : $x - y$ si $x \geq y$; 0 sinon) ;
7. $\lambda xy.|x - y|$;
8. Montrer que la fonction $g = \lambda x, y.2^x + y$ est récursive primitive en la construisant à partir de fonctions de base avec les constructeurs Comp et Rec_Pri.

Solution: Soit $\mathbf{plus} = \lambda xy.x + y$ et $\mathbf{exp2} = \lambda z.2^z$

(a) **plus** est RP :

$$\begin{cases} \mathbf{plus}(x,0) & = & x & = & \pi_1^1(x) \\ \mathbf{plus}(x,n+1) & = & \mathbf{plus}(x,n)+1 & = & \sigma(\mathbf{plus}(x,n)) = h(x,n,\mathbf{plus}(x,n)) \end{cases}$$

Donc $h = \mathbf{Comp}(\sigma, \pi_3^3)$ et $\mathbf{plus} = \mathbf{Rec\_Pri}(\pi_1^1, \mathbf{Comp}(\sigma, \pi_3^3))$

(b) **exp2** est RP :

$$\begin{cases} \mathbf{exp2}(0) & = & 1 & = & \sigma(0) \\ \mathbf{exp2}(n+1) & = & 2^{n+1} & = & \mathbf{exp2}(n) + \mathbf{exp2}(n) = h(n, \mathbf{exp2}(n)) \end{cases}$$

Donc $h = \mathbf{Comp}(\mathbf{plus}, \pi_2^2, \pi_2^2)$
et $\mathbf{exp2} = \mathbf{Rec\_Pri}(\mathbf{Comp}(\sigma, 0), \mathbf{Comp}(\mathbf{plus}, \pi_2^2, \pi_2^2))$

(c) Finalement, $g = \mathbf{Comp}(\mathbf{plus}, \mathbf{Comp}(\mathbf{exp2}, \pi_1^2), \pi_2^2)$

1.1.3 Réaliser une fonction RP en un langage "actionnel"

Programmer les fonctions de base en un langage de programmation comme Pascal est trivial. En ce qui concerne la fonction *Comp*, si l'on sait programmer $h(x_1, \ldots, x_k)$ et $g_1(\bar{x}), \ldots, g_k(\bar{x})$, on peut alors programmer $h(\bar{x}) = f(g_1(\bar{x}), \ldots, g_k(\bar{x}))$. Pour la fonction *Rec*, un programme pour la calculer est le suivant.

```
fonction u(x, y)
  r ← f(x)
  for i = 0 to y − 1 {
     r ← g(x̄, i, r)
  }
  return r
```

On peut donc voir qu'une fonction RP est programmable en utilisant seulement des boucles *for* imbriqués et les appels des fonctions non-récursifs. Or, le langage RP est équivalent à une partie du langage Pascal seulement avec la boucle *for* et sans appels récursifs. ◇

1.1.4 Quelques propriétés de fermeture de la classe RP

Décrire les fonctions RP par des termes ne contenant que des fonctions de base et constructeurs *Comp* et *Rec_Pri* est une tâche fastidieuse. Nous établirons plusieurs propriétés de fermeture de la classe RP qui nous permettront d'établir la récursivité partielle plus rapidement.

Exemple. Considérons $f(n) = \sum_{i=0}^{n} i!$. Afin de montrer que cette fonction est RP, notons d'abord que $i!$ est RP puisqu'on peut l'exprimer comme

$$\begin{cases} 0! = 1 \\ (i+1)! = i!\ (i+1) \end{cases}$$

Ensuite, f peut être écrite comme

$$\begin{cases} f(0) = 1 \\ f(n+1) = f(n) + (n+1)! \end{cases}$$

La fonction f est donc RP. ◇

Proposition 1 *Si $g(\bar{x}, i)$ est RP, alors $f(\bar{x}, n) = \sum_{i=0}^{n} g(\bar{x}, i)$ et $h(\bar{x}, n) = \prod_{i=0}^{n} g(\bar{x}, i)$ sont aussi RP.*

Autrement dit, la classe de fonctions RP est *fermée* par rapport à $\sum$ et $\prod$.
Démonstration. Pour la somme $\sum$ c'est un exercice. Pour le produit $\prod$, on peut écrire la fonction h comme suit.

$$\begin{cases} h(\bar{x}, 0) = g(\bar{x}, 0) \\ h(\bar{x}, n+1) = h(\bar{x}, n) \cdot g(\bar{x}, n+1) \end{cases}$$

Puisque g et la multiplication $\cdot$ sont RP, la construction ci-dessus montre que h est aussi RP. ■

1.2 Les prédicats RP

Un prédicat sur $\mathbb{N}^k$ décrit une propriété d'un k-uplet $(x_1, \ldots, x_k)$. Il peut être défini par une fonction $\mathbb{N}^k \to \{vrai, faux\}$ ou bien par un sous-ensemble de $\mathbb{N}^k$.

Exemple 1. Le prédicat $Pair$ sur $\mathbb{N}$. Nous avons $Pair(5) = faux$ et $Pair(6) = vrai$. Le prédicat $Pair$ correspond à l'ensemble $\{0, 2, 4, 6, 8 \ldots\}$.

Exemple 2. Le prédicat $PlusGrand$ sur $\mathbb{N}^2$ est défini comme suit.

$$PlusGrand(x, y) = \begin{cases} vrai & \text{si } x > y, \\ faux & \text{sinon.} \end{cases}$$

Il correspond à un ensemble $\{(1, 0), (2, 0), \ldots, (2, 1), (3, 1), \ldots\}$.

Dans le cadre défini dans l'introduction un prédicat P sur $\mathbb{N}^k$ correspond au problème (U, B) avec $U = \mathbb{N}^k$ et $B = \{\bar{x} | P(\bar{x})\}$.

Définition 5 (Fonction caractéristique) *Soit P un prédicat sur $\mathbb{N}^k$. Sa fonction caractéristique est $\chi_P : \mathbb{N}^k \to \mathbb{N}$ telle que*

$$\chi_P(\bar{x}) = \begin{cases} 1 & si\ P(\bar{x}), \\ 0 & si\ \neg P(\bar{x}). \end{cases}$$

La fonction caractéristique du prédicat $Pair$ est donc

$$\begin{cases} \chi_{Pair}(2n) = 1, \\ \chi_{Pair}(2n+1) = 0. \end{cases}$$

Et celle du prédicat $PlusGrand$

$$\chi_{PlusGrand}(x, y) = \begin{cases} 1 & \text{si } x > y, \\ 0 & \text{sinon.} \end{cases}$$

Définition 6 (Prédicat RP) *Un prédicat P sur $\mathbb{N}^k$ est RP ssi χ_P est RP.*

Exemple 1. La fonction caractéristique d'un prédicat P tel que $P(x) \Leftrightarrow x \neq 0$ est :

$$\chi_P(x) = \begin{cases} 1 & \text{si } x > 0, \\ 0 & \text{si } x = 0. \end{cases}$$

On peut voir que $\chi_P(x) = sg(x)$ et χ_P est donc RP. Par conséquent, le prédicat P est RP. ◇

Proposition 2 *Soient P et Q deux prédicats RP, alors $P \wedge Q$, $P \vee Q$, $\neg P$ et $P \Rightarrow Q$ sont aussi RP.*

Autrement dit, la classe de prédicats RP est *fermée* par les opérations booléennes.

Démonstration. Nous prouvons d'abord que $P \wedge Q$ est RP. Comme $(P \wedge Q)(\bar{x}) \equiv P(\bar{x}) \wedge Q(\bar{x})$, alors,

$$\chi_{P \wedge Q}(\bar{x}) = \begin{cases} 1 & \text{si } \chi_P(\bar{x}) = 1 \text{ et } \chi_Q(\bar{x}) = 1, \\ 0 & \text{sinon.} \end{cases}$$

On peut en déduire que $\chi_{P\wedge Q}(\bar{x}) = \chi_P(\bar{x}).\chi_Q(\bar{x})$. Or χ_P et χ_Q sont RP par hypothèse et, de plus, on sait que la multiplication est RP. Par conséquent, $\chi_{P\wedge Q}$ est RP et $P \wedge Q$ l'est aussi.

Nous allons maintenant prouver que $\neg P$ est RP.

$$\chi_{\neg P}(\bar{x}) = \begin{cases} 1 & \text{si } \chi_P(\bar{x}) = 0, \\ 0 & \text{si } \chi_P(\bar{x}) = 1. \end{cases}$$

Il est facile de voir que $\chi_{\neg P}(x) = 1 \dot{-} \chi_P(\bar{x})$. Alors, $\chi_{\neg P}$ est RP et $\neg P$ est donc RP.

Pour "$\vee$" et "$\Rightarrow$", on peut les exprimer en utilisant "$\wedge$" et "$\neg$" comme suit : $(P \vee Q) \equiv (\neg(\neg P \wedge \neg Q))$ et $(P \Rightarrow Q) \equiv (\neg(P \wedge \neg Q))$. ■

Par la suite, nous présentons les propriétés de la RP.

Proposition 3 (Quantification bornée) *Soit $P(\bar{x}, y)$ un prédicat RP. Alors, $\forall x \leq n\ P(\bar{x}, y)$ et $\exists y \leq n\ P(\bar{x}, y)$ sont RP.*

Il est important de noter que cette propriété n'est pas vraie pour une quantification arbitraire, par exemple $\forall x P(\bar{x}, y)$ ou bien $\exists y P(\bar{x}, y)$.

Démonstration. Pour $\forall^{\leq}$, notons $Q(\bar{x}, n) \stackrel{\triangle}{=} \forall x \leq n : P(\bar{x}, y)$. Donc,

$$\chi_Q(\bar{x}, n) = \begin{cases} 1 & \text{si } \forall y \leq n : \chi_P(\bar{x}, y) = 1, \\ 0 & \text{sinon.} \end{cases} = \prod_{y=0}^{n} \chi_P(\bar{x}, y)$$

On sait que χ_P est RP, par hypothèse. En plus, selon la Proposition 1, le produit $\prod_{y=0}^{n}$ préserve la récursivité primitive. Ainsi, RP est fermé par $\forall^{\leq}$.

Pour $\exists^{\leq}$, il suffit d'utiliser la propriété que $\exists x \leq n\ P(\bar{x}, y)$ peut être exprimé comme $\exists x \leq n\ P(\bar{x}, y) \equiv \neg(\forall x \leq n : \neg P(\bar{x}, y))$. ■

Proposition 4 (Définition par cas) *Si les fonctions g_i et les prédicats P_i sont RP (pour $i \in \{1, \ldots, n\}$, alors la fonction*

$$f(\bar{x}) = \begin{cases} g_1(\bar{x}) & \textit{si } P_1(\bar{x}), \\ \vdots & \\ g_n(\bar{x}) & \textit{si } P_n(\bar{x}). \end{cases}$$

est aussi RP.

Pour prouver la proposition, notons que la fonction f peut être écrite comme $f = g_1.\chi_{P_1} + \ldots + g_n.\chi_{P_n}$. Alors, f est RP puisqu'elle est obtenue à partir des fonctions RP en utilisant l'addition et la multiplication.

Définition 7 (Opérateur de minimisation bornée) *Soit $P(\bar{n}, m)$ un prédicat. Alors,*

$$f(\bar{n}, m) = \mu^{i \leq m} P(\bar{n}, i) \stackrel{\triangle}{=} \begin{cases} \textit{le plus petit } i \textit{ t.q. } P(\bar{n}, i) & \textit{s'il existe,} \\ 0 & \textit{si un tel } i \textit{ n'existe pas.} \end{cases}$$

On dit que f est obtenue de P par la minimisation bornée.

Proposition 5 (Fermeture de RP par minimisation bornée) *Si P est RP, et f est obtenue de P par la minimisation bornée, alors f est aussi RP*

Démonstration. Soit $f(\bar{n}, m) = \mu^{i \leq m} P(\bar{n}, i)$.

On peut définir la fonction f par cas :

$$f(\bar{n}, m) = \begin{cases} 0 & \text{si } P(\bar{n}, 0) \\ f_1(\bar{n}, m) & \text{sinon.} \end{cases}$$

La fonction f_1 (qui correspond au cas non-trivial de la minimisation quand $P(\bar{n}, 0)$ est faux) peut être définie par récurrence primitive :

$$\begin{cases} f_1(\bar{n}, 0) = & 0 \\ f_1(\bar{n}, m+1) = & \begin{cases} f_1(\bar{n}, m) & \text{si } f_1(\bar{n}, m) > 0 \\ m+1 & \text{si } f_1(\bar{n}, m) > 0 \wedge P(\bar{n}, m+1) \\ 0 & \text{sinon} \end{cases} \end{cases}$$

La première ligne du cas récursif correspond au cas où le plus petit i satisfaisant la propriété est $\leq m$ (et donc nous l'avons déjà trouvé), la deuxième correspond au cas où le plus petit i est $m+1$ (et donc nous ne l'avions pas encore trouvé aux étapes précédentes, mais $P(\bar{n}, m+1)$ est vrai), la dernière est pour le cas où nous ne trouvons pas le bon i ni avant, ni à l'étape $m+1$.

Toutes les fonctions et prédicats qui interviennent dans la définition de f_1 sont RP, et, par conséquent, f_1 l'est aussi. Donc f est aussi RP. ■

1.2.1 Exercices – des fonctions RP plus compliquées

1. Montrer que les prédicats $x \mid y$, $EstPremier(x)$ et $x \; mod \; y = z$ sont RP.
2. **Nombres parfaits.** *Le nombre x est parfait s'il est égal à la somme de tous ses diviseurs (différents de x). Par exemple :* $6 = 1 + 2 + 3$, $28 = 1 + 2 + 4 + 7 + 14$.
   Montrer que le prédicat *EstParfait* est récursif primitif.

   Solution: Le prédicat *est_parfait*(x) est récursif primitif :

   $$\textit{est\_parfait}(x) = \left[x = \sum_{i=1}^{x-1} i \cdot \chi_{|}(i, x) \right]$$

   où $\chi_{|}$ est la fonction caractéristique du prédicat "divise".
3. Montrer que les fonctions suivantes sont récursives primitives en utilisant les propriétés de fermeture :
   (a) $x \bmod y$;
   (b) $x \; \texttt{div} \; y$;
   (c) $ex2(x)$: l'exposant de 2 dans la décomposition de x en facteurs premiers ;
   (d) p_x : le x-ème nombre premier ;

   Solution: On définit la fonction p_n par récursion primitive

   $$\begin{cases} p_0 & = & 2 \\ p_{n+1} & = & \mu x < (n+2)^2 . x > p_n \wedge \forall y < x (y = 1 \vee \neg y | x) \end{cases}$$

   Donc p_n est récursive primitive[a]

   [a] On a utilisé l'inégalité $p_n < (n+1)^2$.

   (e) $ex(x, y)$: l'exposant de p_x dans la décomposition de y en facteurs premiers.
   (f) C_x^y : le nombre de combinaisons de x éléments y à y ;
   (g) $\lfloor \sqrt{\texttt{x}} \rfloor$: la partie entière de $\sqrt{x}$;
   (h) $\lfloor \log_{\texttt{x}} \texttt{y} \rfloor$: la partie entière de $\log_x y$;
4. Les nombres de Fibonacci sont définis par la récurrence :

   $$\begin{cases} Fib(0) & = & 1 \\ Fib(1) & = & 1 \\ Fib(n+2) & = & Fib(n) + Fib(n+1) \end{cases}$$

   Montrer que la fonction Fib est RP.

1.3 Les fonctions RP ne suffisent pas

Nous avons vu dans que la classe de fonctions RP est assez riche pour définir une grande majorité des algorithmes que vous connaissez. En choisissant une bonne représentation de structures de données par des entiers (les détails d'une telle *arithmétisation* sont pareils que dans 1.6.2 ci-dessus) il est possible de représenter comme des fonctions RP les solutions de quasiment tous les problèmes vus en cours de Langages et Programmation. Néanmoins, les deux exemples suivants montrent qu'il existe des fonctions calculables dans le sens informel (par exemple on peut les programmer en Pascal, CAML ou ADA), mais qui ne sont pas récursives primitives. Ceci montre, que malgré tout l'intérêt de la notion de fonctions RP, cette notion *n'est pas une formalisation adéquate de la notion d'algorithme*.

1.3.1 Fonction d'Ackermann

On considère une famille de fonctions définie comme suit.

$$n \uparrow^0 m = n \cdot m = \begin{cases} n \cdot 0 = 0 \\ n \cdot (m+1) = n + n \cdot m \end{cases} \tag{1.1}$$

$$n \uparrow^1 m = \begin{cases} n \uparrow 0 = 1 \\ n \uparrow (m+1) = n \cdot (n \uparrow m) \end{cases} \tag{1.2}$$

$$n \uparrow^2 m = \begin{cases} n \uparrow\uparrow 0 = 1 \\ n \uparrow\uparrow (m+1) = n \uparrow (n \uparrow\uparrow m) \end{cases} \tag{1.3}$$

$$n \uparrow^{k+1} m = \begin{cases} n \uparrow^{k+1} 0 = 1 \\ n \uparrow^{k+1} (m+1) = n \uparrow^k (n \uparrow^{k+1} m) \end{cases} \tag{1.4}$$

Pour calculer $n \uparrow^0 m$ il faut 2 boucles *for* ; 3 boucles *for* pour $n \uparrow^1 m$; 4 boucles *for* pour $n \uparrow^2 m$, ..., $(k+2)$ boucles *for* pour $n \uparrow^k m$.

La fonction d'Ackermann peut être définie comme $Ack(n,k,m) = n \uparrow^k m$.

Pour chaque k fixé, la fonction $\uparrow^k$, c'est-à-dire $\lambda nm.n \uparrow^k m$ est RP. Pourtant, on peut montrer que la fonction $Ack = \lambda nkm.n \uparrow^k m$, k étant une variable, n'est pas RP. Nous omettrons la preuve basée sur le fait que la fonction Ack croît plus vite que toutes les fonctions récursives primitives.

Par contre, on peut facilement écrire un programme récursif qui termine toujours et qui calcule Ack (il faut juste programmer les équations de récurrence 1.1- 1.4).

Nous venons donc de trouver une fonction calculable (dans le sens informel) qui n'est pas RP.

1.3.2 Interpréteur RP

Un autre exemple important d'une fonction calculable non RP est l'interpréteur des termes RP (ou la fonction universelle). Pour démontrer que cet interpréteur n'est pas RP nous utilisons (pour la première fois dans ce cours) la méthode de "diagonalisation", qui jouera un rôle important dans la suite.

Énumération La première idée est d'énumérer toutes les fonctions RP de forme $\mathbb{N} \to \mathbb{N}$:

$$f_0(x), f_1(x), \ldots, f_i(x) \ldots \tag{1.5}$$

Pour obtenir une telle énumération nous pouvons remarquer, que chaque fonction RP f peut être définie par un terme contenant les fonctions de base et les constructeurs de la classe RP. Par exemple, la fonction $doubler = \lambda x.2x$ est définie par

$$doubler \stackrel{\triangle}{=} Rec\_Pri(0, Comp(\sigma, Comp(\sigma, \pi_2^2))),$$

c'est-à-dire par une chaîne de caractères ASCII. On peut énumérer toutes les chaînes en ordre lexicographique. La liste (1.5) contient toutes les fonctions RP et seulement les fonctions RP. Nous allons noter la chaîne numéro i par s_i.

Nous définissons ensuite un interpréteur RP (une fonction universelle $Int(i,x)$ où i est le numéro du programme et x est l'entrée) comme suit :

$$Int(i,x) = \begin{cases} g(x) & \text{si } s_i \text{ est la déf. d'une fonction RP } g : \mathbb{N} \to \mathbb{N}, \\ 0 & \text{si } s_i \text{ n'est pas une telle définition.} \end{cases}$$

Nous pouvons remarquer que Int est calculable (dans le sens usuel). Un programme pour calculer $Int(i,x)$ devrait générer d'abord la chaîne s_i, effectuer son analyse syntaxique. Si s_i n'est pas un terme RP pour une fonction scalaire, Int renvoie 0, sinon Int utilise l'arbre syntaxique obtenu et les définitions des opérations $Comp$ et rec_pri pour calculer la valeur de $f_i(x)$

Finalement on obtient la liste de toutes les fonctions RP en posant $f_i(x) = Int(i,x)$. Par construction chaque fonction f_i est soit nulle, soit représentée par un terme récursif primitif, et donc, dans les deux cas elle est RP. Réciproquement, chaque fonction RP f admet un terme t qui se représente par une chaîne s qui a un numéro lexicographique i, donc f est présente dans la liste sous ce numéro i.

Diagonalisation

Proposition 6 *La fonction Int n'est pas RP.*

Démonstration (par contradiction). Supposons que Int est RP. Définissons $h(x) \stackrel{\triangle}{=} f_x(x) + 1 = Int(x,x) + 1$. Cette fonction doit aussi être RP (comme Int l'est). Donc h est dans la liste 1.5, c'est-à-dire $h = f_i$ pour un certain i. Comme $\forall x h(x) = f_i(x)$, et avec $x = i$ nous avons $h(i) = f_i(i)$. Or, par définition, $h(i) = f_i(i) + 1$. On obtient donc $f_i(i) = f_i(i) + 1$ – contradiction !

Nous en déduisons que notre hypothèse de départ était fausse et Int est RP (h non plus). ■

On peut re-formuler la même preuve en terme suivants. Nous définissons h comme suit :

$$\begin{aligned} h(1) &= f_1(1) + 1 \\ h(2) &= f_2(2) + 1 \\ &\vdots \\ h(k) &= f_k(k) + 1 \end{aligned}$$

Il est clair que pour chaque $x = k$, $h(k) \neq f_k(k)$, ce qui implique que $h \neq f_k$ pour tout k. Donc h n'est pas dans la liste de toutes les fonctions RP, et donc h n'est pas RP. ■

1.4 Fonctions récursives partielles

Nous avons vu que la classe de fonctions RP est en fait une sous-classe de fonctions calculables. La question qui se pose ensuite est comment construire la vraie classe de fonctions calculables. Même si l'on élargit la classe de fonctions de base et les opérations, un contre-exemple similaire à la fonction "diagonale" h ci-dessus est toujours valide. En effet, soit $f_0(x), f_1(x), \ldots, f_i(x) \ldots$ la liste complète de fonctions calculables telle que $f_i(x) = Int(i,x)$. Alors, $h(x) = f_x(x) + 1$ est une fonction calculable qui n'est pas dans la liste. La solution à ce problème est de considérer les fonctions partielles.

Définition 8 (Fonction partielle) *Une fonction* partielle $f : \mathbb{N}^k \cdots \to \mathbb{N}$ *est une fonction d'un sous-ensemble de* $\mathbb{N}^k$ *vers* $\mathbb{N}$. *Le domaine de* f *est* $Dom(f) = \{\bar{x} \mid f(\bar{x}) \text{ est définie}\}$. *Si en* $\bar{x}$ *la fonction* f *est définie, on écrit* $f(\bar{x}) \downarrow$ *(on lit :* f *converge sur* $\bar{x}$*), sinon on écrit* $f(\bar{x}) \uparrow$ *ou* $f(\bar{x}) = \perp$ *(*f *diverge). Si* $Dom(f) = \mathbb{N}^k$ *on dit que* f *est totale*[1]

Par convention, quand on applique des opérations arithmétiques (ou d'autres fonctions) à une valeur indéfinie $\perp$, le résultat diverge également.

[1] Des fonctions totales forment une sous-classe des fonctions partielles.

Exemple 1.

$$f(x) = \begin{cases} x/2 & \text{si } x \text{ est pair,} \\ \uparrow & \text{sinon.} \end{cases}$$

Alors, $f(5) + 1 \uparrow$, $f(6) + 1 = 3 + 1 = 4$, et $0 \cdot f(7) + 2 \uparrow$. En général, si f est une fonction partielle, nous avons

$$0 \cdot f(x) = \begin{cases} 0 & \text{si } f(x) \downarrow \\ \uparrow & \text{si } f(x) \uparrow \end{cases}$$

1.4.1 Minimisation non-bornée

Afin de définir des fonctions récursives partielles, nous allons introduire la définition de la *minimisation non-bornée*. Cette opération ressemble à la minimisation bornée, voir définition 7.

Définition 9 (provisoire) *Si $f(\bar{x}, y)$ est totale, on peut définir la minimisation non-bornée comme suit.*

$$g(\bar{x}) = \mu y.(f(\bar{x}, y) = 1) = \begin{cases} \textit{le plus petit } y \textit{ tel que } f(\bar{x}, y) = 1, \\ \uparrow \textit{ s'il n'y a pas de tel } y. \end{cases} \tag{1.6}$$

Exemples

$$\mu y.((y+1)^2 > x) = \lfloor \sqrt{x} \rfloor$$

$$\mu y.(y^2 = x) = \begin{cases} \sqrt{x} & \text{si } x \text{ est un carré parfait,} \\ \uparrow & \text{sinon.} \end{cases}$$

$$\mu k.(2k = x) = \begin{cases} x/2 & \text{si } x \text{ est pair,} \\ \uparrow & \text{sinon.} \end{cases}$$

On voit que la minimisation bornée fait sortir de la classe des fonctions totales.

Au niveau toujours informel on peut se convaincre que le résultat de la minimisation est calculable

Proposition 7 *Si f est calculable totale, alors la minimisation non-bornée g définie en (1.6) est aussi calculable.*

Démonstration. Afin de prouver la proposition, nous montrons le programme suivant qui calcule g. Notons que pour programmer la minimisation non-bornée, il faut utiliser des boucles *while*.

fonction $g(\bar{x})$
  $y \leftarrow 0$
  while $(f(\bar{x}, y) \neq 1)$ {
    $y \leftarrow y + 1$
  }
  return y

■

Maintenant nous allons considérer le cas où f est partielle. On peut voir que la définition (1.6) n'est plus adéquate puisque le programme qui calcule $\mu y.f(\bar{x}, y)$ boucle si $f(\bar{x}, y)$ n'est pas définie pour certain y. Ainsi, nous allons donner ensuite une définition plus rigoureuse.

Définition 10 *Soit $f(\bar{x}, y)$ est fonction (peut-être partielle), la minimisation non-bornée est définie comme suit.*

$$\begin{aligned} g(\bar{x}) &= \mu y.(f(\bar{x}, y) = 1) \\ &= \begin{cases} \textit{le } y \textit{ tel que} & f(\bar{x}, y) = 1 \wedge \forall i < y(f(\bar{x}, i) \downarrow \wedge f(\bar{x}, i) \neq 1) \\ \uparrow & \textit{s'il n'y a pas de tel } y. \end{cases} \end{aligned}$$

On voit donc, que $\mu y.(f(\bar{x}, y) = 1)$ est le plus petit y convenable, à condition que pour toutes les valeurs plus petites la fonction est quand même définie.

Remarque. Il est très facile de se tromper en appliquant μ à des fonctions qui ne sont pas totales. C'est une pratique déconseillée (et inutile, comme nous démontrerons dans la suite).

1.4.2 Fonctions récursives partielles et la thèse de Church

Définition 11 (Fonctions récursives partielles) *La classe de fonctions récursives partielles est la plus petite classe de fonctions partielles qui contient les fonctions de base* $0, \pi, \sigma$*, et qui est fermée par les constructeurs Comp, Rec_Pri et* μ*.*

La seule différence par rapport à la définition de la classe RP est le nouveau constructeur μ. C'est trivial que toutes les fonctions récursives primitives RP sont aussi récursives partielles.

Un lecteur attentif peut remarquer ici, que pour donner un sens précis à cette définition, il faut étendre les opérations *Comp*, *Rec_Pri* aux fonctions partielles. C'est un exercice pour ce lecteur.

Thèse 1 (Thèse de Church) *La classe de fonctions récursives partielles est égale à la classe de fonctions calculables par un algorithme quelconque.*

Notons que ceci est une thèse (et pas un théorème) puisque la classe de fonctions récursives partielles est formellement définie tandis que "la classe de fonctions calculables" est une notion intuitive et informelle.

La thèse de Church contient deux parts :

1. Chaque fonction récursive partielle est calculable.
2. Chaque fonction calculable est récursive partielle.

Le raisonnement suivant permet de se convaincre de la première partie de la thèse de Church

"Démonstration." Afin de prouver la première part, nous allons montrer que les fonctions de base sont calculables, aussi bien que les constructeurs (qui sont utilisés pour définir les fonctions récursives partielles) préservent la calculabilité, et il suffit de montrer les programmes qui calculent ces fonctions. Pour les fonctions de base, c'est trivial de montrer les programmes qui les calculent. En ce qui concerne les constructeurs, supposons que nous avons les algorithmes pour calculer f et g, les programmes pour calculer les résultats d'application des constructeurs sont montrés ci-dessous.

fonction $Comp(f, g)(\bar{x})$
  return $f(g(\bar{x}))$

fonction $Rec\_Pri(f, g)(\bar{x}, y)$
  $r \leftarrow f(\bar{x})$
  for $i = 1$ **to** y {
    $r \leftarrow g(\bar{x}, i, r)$
  }
  return r

fonction $\mu f = 1(\bar{x})$
  $k \leftarrow 0$
  while $f(\bar{x}, k) \neq 1$ {
    $k \leftarrow k + 1$
  }
  return k

■

Il existent aussi des arguments en faveur de la réciproque. En particulier toutes les fonctions calculable par des algorithmes qu'on peut représenter en langages de programmations qui existent

actuellement sont récursives partielles. Des nombreuses définitions théoriques des fonctions calculables sont équivalentes à la récursivité partielle. Donc si la thèse de Church est fausse et s'il existe un algorithme qui n'est pas récursif partiel, cet algorithme doit être basé sur des principes actuellement inconnus. Ceci me semble très peu probable.

NOUS AVONS FINALEMENT RÉUSSI À TROUVER UNE NOTION FORMELLE DE FONCTION CALCULABLE : C'EST LA NOTION DE FONCTION RÉCURSIVE PARTIELLE.

1.4.3 Exercices – fonctions récursives partielles

1. Montrer que les fonctions suivantes sont récursives partielles (en utilisant l'opérateur de minimisation) : $x - y$; $\frac{x}{y}$; $\log_x y$ (elles sont définies seulement sur x et y tels que le résultat est naturel).
2. Montrer que les fonctions suivantes sont récursives partielles :
   (a)
   $$f(x)^2 = \begin{cases} \text{une solution de l'équation } y^3 - 5y^2 = 3 & \text{si elle existe} \\ \uparrow & \text{s'il n'y a pas de solution} \end{cases}$$
   (b)
   $$f(x) = \begin{cases} 1 & \text{si } x \text{ peut être représenté comme } y^2 + z^2 \\ \uparrow & \text{sinon} \end{cases}$$
   (c)
   $$f(n)^3 = \begin{cases} 1 & \text{si } \exists x, y, z \cdot x^n + y^n = z^n \\ \uparrow & \text{sinon} \end{cases}$$
3. Montrer que le programme suivant calcule une fonction récursive partielle (problème de Collatz) :

```
function f(x)
  n := 0;
  while x > 1
        n := n + 1
        if pair(x)
          then x := x div 2
          else x := 3x + 1
  return n
```

1.5 Machines de Turing

Nous allons maintenant étudier un autre modèle de calcul qui est la machine de Turing. Ce modèle est très différent du modèle fonctionnel des fonctions récursives partielle. Il ressemble plus aux modèles de calcul de type "machine", tels que des automates ou des machines à pile (et aux ordinateurs réels). La raison pour laquelle nous concentrons sur les machines de Turing est qu'elles sont des machines abstraites très simples mais capables de réaliser *tous* les algorithmes.

Pour mieux placer les machines de Turing dans le contexte nous pouvons considérer d'abord des modèles plus simples. Un automate fini, n'ayant pas de "mémoire externe", a une capacité de calcul assez limitée. Une machine à pile est un automate avec une pile, qui représente une mémoire externe non-bornée, mais avec l'accès très restreint (seulement par le sommet). Une machine de Turing peut être vue comme un automate avec un ruban de longueur infinie (qui est une mémoire avec des règles d'accès assez libérales). Plus précisément, chaque case du ruban contient un symbole (on suppose que seulement un nombre fini de cases contiennent un symbole non-nul), la machine possède une tête qui peut se déplacer sur le ruban, lire le symbole à sa position, et réécrire ce symbole. La définition formelle suit.

[2] cas particulier du 10-ème problème de Hilbert.
[3] Problème de Fermat.

Définition 12 (Machine de Turing) *Une machine de Turing (une MT) est un quadruplet* $MT = (Q, \Sigma, q_0, \delta)$ *tel que*

- Q *: un ensemble fini appelé l'ensemble d'états,*
- Σ *: un alphabet fini appelé l'alphabet du ruban (on suppose que cet alphabet contient 0 et 1),*
- q_0 *: un élément de* Q *appelé l'état initial,*
- δ *: le programme qui est un ensemble d'instructions.*

Les instructions ont la forme suivante :(état, symbole lu) $\rightarrow$ *(état, symbole écrit, déplacement) où les deux états appartiennent à* Q*, le deux symboles à* Σ*, et le déplacement à* $\{G, D, -\}$*. Il y a donc trois types d'instructions :*

- $(q, a) \rightarrow (p, b, G)$ *(en observant* a *en état* q *passer à l'état* p*, écrire* b*, déplacer la tête à gauche),*
- $(q, a) \rightarrow (p, b, D)$ *(déplacer à droite),*
- $(q, a) \rightarrow (p, b, -)$ *(aucun déplacement).*

Définition 13 ***Configurations et calculs d'une machine de Turing***

- *Une configuration d'une MT est* (w_G, q, w_D) *où* w_G *est le mot à gauche de la tête de lecture (jusqu'au premier symbole non-nul),* q *est l'état actuel,* w_D *est le mot à droite de la tête de lecture (y compris la case où la tête se trouve, et jusqu'au dernier symbole non-nul).*
- *Un calcul de la machine sur une entrée* $(x_1, \ldots, x_k) \in \mathbb{N}^k$ *est une séquence de configurations* $c_1, c_2, \ldots, c_n$ *telle que*
  1. *Chaque configuration* c_{i+1} *est obtenue à partir de la configuration* c_i *en appliquant une instruction de la machine*[4]*.*
  2. $c_1 = \varepsilon, q_0, 01^{x_1}01^{x_1}\ldots01^{x_k}$*.*
  3. *Dans la configuration* c_n *il n'y a pas d'instructions applicables.*

Dorénavant, nous considérons seulement les machines de Turing *déterministes*, c'est-à-dire les machines ayant au plus une instruction commençant par le même (q, a).

Exemple. Supposons que la machine M a seulement 2 instructions :

$$\begin{array}{ll} I_1: & (q_0, 0) \rightarrow (q_1, 1, D) \\ I_2: & (q_1, 0) \rightarrow (q_1, 3, G) \end{array}$$

et qu'initialement le ruban de la machine M ne contient que des 0' :

$$\begin{array}{ccccccccccccc} \ldots & | & 0 & | & 0 & | & 0 & | & 0 & | & 0 & | & 0 & | \ldots \\ & & & & & & \triangle & & & & & & \end{array} .$$

Alors, après avoir exécuté le programme, la machine est dans la configuration suivante :

$$\begin{array}{ccccccccccccc} \ldots & | & 0 & | & 0 & | & 1 & | & 3 & | & 0 & | & 0 & | \ldots \\ & & & & & & \triangle & & & & & & \\ & & & & & & q_0 & & & & & & \end{array}$$

Le calcul de la machine est donc $(\varepsilon, q_0, \varepsilon) \rightarrow (1, q_1, \varepsilon) \rightarrow (\varepsilon, q_0, 13)$. $\diamond$

Définition 14 (Fonctions calculées par les machines de Turing) *Le résultat de calcul de la machine* M *est le nombre de "*1*" sur le ruban quand la machine est dans la configuration* c_n*. La fonction(partielle)* $f_M^{(k)}$ *de* k *arguments calculée par la machine* M *est donc définie comme suit :*

- *Si sur l'entrée* $(x_1, \ldots, x_k)$ *la machine* M *s'arrête dans une configuration* c_n *alors* $f(x_1, \ldots, x_k) =$ *nombre de "*1*" dans* c_n*.*
- *si elle ne s'arrête pas, alors* $f(x_1, \ldots, x_k) \uparrow$

[4]pour être rigoureux il faudrait formaliser cela.

Exemples. Voir exercices 1-2, section 1.6.1.

1.6 Thèse de Church-Turing. Équivalence MT – fonctions récursives partielles

Nous présentons maintenant les résultats importants qui montrent le lien entre les fonctions calculées par les machines de Turing et les fonctions récursives partielles.

Théorème 1 *Chaque fonction récursive partielle peut être calculée par une machine de Turing.*

Théorème 2 *Chaque fonction calculée par une machine de Turing est récursive partielle.*

D'où la forme équivalente de la thèse de Church

Thèse 2 (Thèse de Church-Turing.) *La classe de fonctions calculables par machines de Turing est égale à la classe de fonctions calculables par un algorithme quelconque.*

Pour prouver les deux théorèmes ci-dessus, nous construirons deux compilateurs : l'un qui compile une fonction récursive partielle vers une machine de Turing, et l'autre qui compile une machine de Turing vers une fonction récursive partielle. Nous présenterons dans la suite ces preuves avec plus de détails.

Preuve du théorème 1

Nous allons montrer d'abord quelques résultats préliminaires qui seront utilisés pour la preuve. En outre, pour des raisons techniques, nous aurons besoin de calculs *réguliers*. Une machine M calcule f régulièrement si elle satisfait les deux conditions suivantes :

1. Quand la machine M s'arrête, elle le fait toujours dans une configuration de la forme

$$\ldots \mid 0 \mid \underset{\substack{\triangle \\ q_1}}{0} \mid 1^{f(\bar{x})} \mid 0 \mid \ldots$$

2. Elle ne va jamais à gauche de sa position initiale.

Nous prouverons par induction que pour chaque fonction récursive partielle il existe une MT qui la calcule régulièrement.

Le cas de base est laissé comme exercice (voir section 1.6.1) :

Lemme 1 *Il existe des machines de Turing qui calculent régulièrement les fonctions* $0()$, $\sigma(x)$ *et* $\pi_i^k(x_1, \ldots, x_k)$.

Pour le cas inductif il faut considérer les trois constructeurs. Nous donnerons une preuve pour *Comp* et nous donnerons des exercices pour illustrer les deux autres preuves.

Lemme 2 *Si* f *et* $g_1, \ldots, g_k$ *sont calculables par une MT, alors* $Comp(f, g_1, \ldots, g_k)$ *est calculable par une MT.*

Démonstration. Nous considérons uniquement le cas de deux fonctions scalaires. Soient f et g deux fonctions MT-calculables : $f = f_R^{(1)}$ (calculable par la machine R), $g = f_S^{(1)}$ (calculable par la machine S). Nous voulons alors calculer $f(g(x))$.

Soient $r_0, r_1, \ldots, r_n$ les états de R, et $s_0, s1, \ldots, s_m$ les états de S (nous supposons ces deux ensembles disjoints).

Nous allons construire une machine de Turing M pour calculer $f(g(x))$. Les états de M sont

$$q_0, q_1, r_0, r_1, \ldots, r_n, s_0, s1, \ldots, s_m$$

et ses instructions sont

$$I_1 : q_0 0 \to s_0 0 \text{ (lancer } S),$$
$$\delta_S \text{ (instructions de } S),$$
$$I_2 : s_1 0 \to r_0 0 \text{ (lancer } R),$$
$$\delta_R \text{ (instructions de } R),$$
$$I_3 : r_1 0 \to q_1 0 \text{ (\textbf{stop})}.$$

Les étapes de calcul de M sont

$$(\varepsilon, q_0, 01^x) \xrightarrow{I_1} (\varepsilon, s_0, 01^x) \xrightarrow{S} (\varepsilon, s_1, 01^{g(x)}) \xrightarrow{I_2} (\varepsilon, r_0, 01^{g(x)}) \xrightarrow{R} (\varepsilon, r_1, 01^{f(g(x))}) \xrightarrow{I_3} (\varepsilon, q_1, 01^{f(g(x))}),$$

ce qui conclut la preuve. ■

Lemme 3 *Si f et g sont calculables par une MT, alors $Rec\_Pri(f, g)$ est calculable par une MT.*

Voir exercice 5a ci-dessous pour une idée de preuve.

Lemme 4 *Si f est calculable par une MT, alors $\mu y.f(\bar{x}, y) = 1$ est calculable par une MT.*

Voir exercice 5b ci-dessous pour une idée de preuve.

Le théorème 1 est une conséquence immédiate des lemmes ci-dessus et de la définition des fonctions récursives partielles. Nous venons de montrer que "récursive partielle ⇒ MT-calculable régulièrement". Cette démonstration est constructive et peut être vue comme une description d'un compilateur des fonctions récursives partielles vers des MT (régulières) ◇

1.6.1 Exercices – MT

1. Quelle fonction $f(x)$ (d'un seul argument) est calculée par cette Machine de Turing ?

$$\begin{aligned} q_0 0 &\longrightarrow q_2 0 D \\ q_0 1 &\longrightarrow q_1 1 \\ q_2 0 &\longrightarrow q_1 1 \\ q_2 1 &\longrightarrow q_2 1 D \end{aligned}$$

2. Quelles fonctions sont calculées par cette Machine de Turing (donner toutes les fonctions de 0, 1, 2 et plus arguments) ?

$$q_0 0 \longrightarrow q_1 0$$

3. Écrire les machines de Turing qui calculent (régulièrement) les fonctions suivantes :
   (a) $0()$;
   (b) $\sigma(x)$;
   (c) $\pi_2^3(x, y, z)$.
4. Écrire les machine de Turing qui font les transformations suivantes :

$$(\varepsilon, q_0, 01^x) \longrightarrow (\varepsilon, q_1, 01^x 01^x) \tag{1.7}$$

$$(1^x, q_0, 01^y) \longrightarrow (1^y, q_1, 01^x). \tag{1.8}$$

5. Définir les machines de Turing qui calculent (régulièrement) :
   (a) $x \cdot y$;
   (b) $\sqrt{x}$.

1.6.2 Vers la preuve de théorème 2

En ce qui concerne le théorème 2, la difficulté de sa preuve est que les machines de Turing fonctionnent sur Σ^* tandis que les fonctions récursives partielles sont définies sur les entiers. Il faut donc représenter les mots ainsi que les séquences de mots par les entiers. Pour ce but, nous allons utiliser *l'arithmérisation* ou *Gödelisation*.

Gödelisation

L'idée est de donner un numéro entier naturel à chaque objet utilisé (mot, instruction, machine de Turing...) et de remplacer des manipulation de ces objets par des opérations de nature arithmétique sur leurs numéros

Nous commençons par les séquences d'entiers. Le numéro de Gödel d'un n-uplet d'entiers naturels $(x_1, \ldots, x_n)$, dénoté par $\langle x_1, \ldots x_n \rangle$, est défini comme suit :

$$A = \langle x_1, \ldots x_n \rangle = 2^n \cdot 3^{x_1} \cdot 5^{x_2} \cdots p_n^{x_n} = 2^n \prod_{i=1}^{n} p_i^{x_i}.$$

(nous utilisons la notation p_i comme notation pour le n-ème nombre premier avec $p_0 = 2$ et $p_1 = 3$)

Par exemple, $\langle 10, 11 \rangle = 2^2 \cdot 3^{10} \cdot 5^{11} = 11533007812500$. Nous mentionnons ensuite quelques opérations usuelles sur les numéros de Gödel :

- **est_un_no**(A) : ce prédicat est vrai ssi A est le numéro d'un uplet
- **long**(A) : longueur de l'uplet (par exemple n pour $A = \langle x_1, \ldots x_n \rangle$). Pour complétude on pose la valeur de cette fonction à 0 si A n'est pas un numéro.
- **el**(A, i) : élément x_i (ou 0 si A n'est pas un numéro, ou si la séquence ne contient pas assez d'éléments)
- **remplacer**(A, i, m) : remplacer x_i par m, c-à-d pour $A = \langle x_1, \ldots, x_i, \ldots, x_n \rangle$, le résultat est **remplacer**$(A, i, m) = \langle x_1, \ldots, m, \ldots, x_n \rangle$.

Propriétés des numéros de Gödel

- Le numéro de Gödel d'un n-uplet est une opération injective (c-à-d des n-uplets différents ont des numéros différents).
- Les opérations énumérées ci-dessus sur les numéros de Gödel (et autres opérations naturelles) sont récursives primitives.

Démonstration. Le numéro de Gödel d'un n-uplet est injective puisque la décomposition en facteurs premiers est unique. La preuve que les opérations sont récursives primitives fait l'objet de l'exercice 1 (section 1.6.3). ■

Arithmétisation de MT

Supposons que l'alphabet d'une machine de Turing M est $\Sigma = \{a_0, a_1, \ldots, a_n\}$ (avec $a_0 = 0$ et $a_1 = 1$) et que ses états sont $Q = \{q_0, \ldots, q_m\}$.

Numéro d'un mot. Soit $w \in \Sigma^*$ tel que $w = a_{i_1} a_{i_2} \ldots a_{i_l}$. Le numéro de w est $\langle w \rangle = \langle i_1, i_2, \ldots i_l \rangle$. Par exemple, $\langle a_3 a_7 a_1 \rangle = \langle 3, 7, 1 \rangle = 2^3.3^3.5^7.7^1$.

Numéro d'une instruction. Étant donné une instruction

$$I : q_i a_j \rightarrow q_k a_l \left\{ \begin{array}{c} - \\ G \\ D \end{array} \right\}$$

$$\langle I \rangle = \langle i, j, k, l \left\{ \begin{array}{c} 0 \\ 1 \\ 2 \end{array} \right\} \rangle$$

Numéro d'une machine de Turing. Soient $I_1, \ldots, I_n$ les instructions de la machine M. Alors,

$$\langle M \rangle = \langle \langle I_1 \rangle, \ldots, \langle I_n \rangle \rangle$$

Numéro d'une configuration. Soit $c = (w_G, q_i, w_D)$ une configuration.

$$\langle c \rangle = \langle \langle w_G \rangle, \langle i \rangle, \langle w_D \rangle \rangle$$

Démonstration de théorème 2

Nous allons le prouver uniquement pour le cas scalaire. Pour établir la preuve, nous définissons les fonctions suivantes :

- **init**(x) donne le numéro de la configuration initiale pour l'entrée x. La fonction **init**(x) est récursive primitive (voir TD).
- **suivant**(m, c) donne le numéro de la configuration de la machine numéro m après une transition à partir de la configuration numéro c. La fonction **suivant**(m, c) est récursive primitive (exercice).
- **conf**(m, x, t) donne le numéro de la configuration de la machine numéro m après t transitions à partir de l'entrée x. On peut prouver que **conf**(m, x, t) est récursive primitive :

$$\begin{cases} \textbf{conf}(m, x, 0) = \textbf{init}(x) \\ \textbf{conf}(m, x, t+1) = \textbf{suivant}(m, \textbf{conf}(m, x, t)) \end{cases}$$

- **stop**(m, c) donne $Vrai$ si la configuration numéro c est une configuration de l'arrêt de la machine numéro m (exercice).
- **temps_de_calcul**$(m, x) = \mu t \,.\, \textbf{stop}(m, \textbf{conf}(m, x, t))$. Cette fonction est aussi récursive partielle.
- **sortie**(c) donne le nombre de "1" sur le ruban dans la configuration numéro c. **sortie**(c) est récursive primitive (voir TD).

Ensuite, en utilisant les fonctions ci-dessus nous pouvons définir la fonction

$$\textbf{la\_fonction}(m, x) = \textbf{sortie}(\textbf{conf}(m, x, \textbf{temps\_de\_calcul}(m, x)))$$

qui donne en fait le résultat du calcul de la machine numéro m sur l'entrée x.

Résumons ce que nous avons établi. Soit $f(x)$ une fonction Turing-calculable, c'est-à-dire il existe une machine de Turing M telle que $f(x) = f_M(x)$. Soit m_0 le numéro de M. Alors $f(x) =$ **la_fonction**(m_0, x), indiquant que f est récursive partielle. Nous venons de prouver que "MT-calculable" $\Rightarrow$ "récursive partielle" (pour le cas de fonctions d'un seul argument). L'extension au cas général est presque triviale. ■

Notons que le but initial étant de chercher un compilateur d'une machine de Turing vers une fonction récursive partielle, nous avons trouvé en fait un interpréteur (c-à-d **la_fonction**(m, x)) – une fonction récursive partielle capable de trouver le résultat de chaque machine de Turing sur chaque entrée. Comme ces résultats ont été établis avant l'invention de la programmation (et des interpréteurs), cette fonction a été appelée la *fonction universelle*. Dans la suite, nous étudions les propriétés de cette fonction.

1.6.3 Exercices – Gödelisation

Cette série d'exercices permet de compléter la preuve ci-dessus

1. Montrez que le prédicat

$$EstUnNumero(M) = \begin{cases} \text{vrai} & \text{s'il existe une représentation} \quad M = \langle x_0 x_1 \ldots x_{n-1} \rangle \\ \text{faux} & \text{sinon} \end{cases}$$

et les fonctions

$$longueur(M) = \begin{cases} n & \text{s'il existe une représentation} \quad M = \langle x_1 x_2 \dots x_n \rangle \\ 0 & \text{sinon} \end{cases}$$

$$element(M, i) = \begin{cases} x_i & \text{s'il existe une représentation} \quad M = \langle x_1 x_1 \dots x_n \rangle, \quad n \geq i > 0 \\ 0 & \text{sinon} \end{cases}$$

sont récursives (primitives).

2. Trouvez la forme explicite de la fonction suivante et déduisez sa récursivité primitive :

$$init(x) = \text{numéro de Gödel de la configuration initiale de la MT pour l'entrée } x.$$

3. Prouvez que la fonction suivante est récursive primitive :

$$sortie(x) = \text{le nombre de 1s sur le ruban dans la configuration de numéro de Gödel } c.$$

1.6.4 Fonction universelle

Nous dénotons la fonction universelle par U, c-à-d $U = \textbf{la\_fonction}$.

Théorème 3 (Théorème d'énumération)

1. *$U(m, x)$ est récursive partielle (Turing-calculable)*
2. *Pour chaque fonction $f(x)$ récursive partielle (ou Turing-calculable) avec un seul argument, il existe m_0 tel que $f(x) = U(m_0, x)$ pour tout x.*

Démonstration. Pour (1), nous avons déjà montré que $U(m, x)$ est récursive partielle dans la preuve du théorème 2. Pour (2), comme $f(x)$ est Turing-calculable, par le théorème 1, il existe une machine de Turing M telle que $f(x) = f_M(x)$. Soit m_0 le numéro de la machine M. En conséquence, $f(x) = U(m_0, x)$. ■

Nous avons présenté le théorème 3 pour le cas scalaire, mais il peut être facilement étendu au cas vectoriel. Pour ce cas-là où $\bar{x} = (x_1, \dots, x_k)$, la fonction universelle a la forme $U^{(k)}(m, \bar{x})$. Nous définissons la fonction $\lambda m, \bar{x} \,.\, \varphi_m^{(k)}(\bar{x}) \stackrel{\triangle}{=} U^{(k)}(m, \bar{x})$, c-à-d $\varphi_m^{(k)}(\bar{x})$ est une fonction récursive partielle de k arguments calculée par la machine numéro m et $\varphi_m^{(k)}(\bar{x})$ est appelée la fonction récursive partielle numéro m de k arguments. Nous avons alors énuméré toutes les fonctions calculables :

Théorème 4 (Théorème d'énumération - forme 2)

1. *La liste*
$$\varphi_0^{(k)}, \varphi_1^{(k)}, \varphi_2^{(k)}, \dots$$
est une énumération de toutes les fonctions récursives partielles de k arguments.
2. *Cette énumération est "effective" (calculable) dans le sens que la fonction $\lambda m, \bar{x} \,.\, \varphi_m^{(k)}(\bar{x})$ est récursive partielle.*

Dans la suite nous utiliserons systématiquement la notation $\varphi_m^{(k)}$ en omettant parfois l'indice d'arité (k).

1.7 Théorème de la forme normale

Une autre conséquence importante de la preuve du théorème 2 est la suivante. Rappelons que

$$U(m, x) = \textbf{sortie}(\textbf{conf}(m, x, \textbf{temps\_de\_calcul}(m, x))).$$

D'ailleurs, $\textbf{temps\_de\_calcul}(m, x) = \mu t \,.\, \textbf{stop}(m, \textbf{conf}(m, x, t))$. La fonction $f_M(x)$ peut être donc écrite sous forme $f_M(x) = g(x, \mu t \,.\, P(x, t))$, ce qui montre que pour exprimer $f_M(x)$, seule l'opérateur de minimisation μ suffit. Nous avons établi le résultat suivant :

Théorème 5 (Théorème de la forme normale) *Chaque fonction récursive partielle peut être représentée comme*

$$f_M(x) = g(x, \mu t \,.\, P(x,t))$$

où g est une fonction récursive primitive et P est un prédicat récursif primitif.

Nous laissons comme exercice le résultat suivant qui donne une forme normale un peu plus simple :

Théorème 6 (Théorème de la forme normale 2) *Chaque fonction récursive partielle peut être représentée en utilisant seulement l'opérateur de minimisation μ :*

$$f_M(x) = h(\mu y \,.\, Q(x,y))$$

où h est une fonction récursive primitive et Q est un prédicat récursif primitif.

Remarque. Une conséquence du théorème 5 est que tout algorithme peut être programmé avec *une* seule boucle *while* et plusieurs boucles *for*.

1.8 Théorème de paramétrisation (s-m-n)

Maintenant nous allons considérer le problème suivant. Étant donné une fonction $f(x,y)$ de 2 arguments, nous voulons traiter x comme paramètre afin d'obtenir une famille de fonctions calculables d'un argument, par exemple,

$$\begin{array}{c} f(0,y) \\ f(1,y) \\ \vdots \\ f(2003,y) \end{array}$$

En d'autres termes, le problème que nous voulons résoudre est formulé comme suit. Supposons que $f(x,y)$ est calculable par une machine de Turing numéro i, c'est-à-dire $f(x,y) = \varphi_i^{(2)}(x,y)$, on veut trouver les programmes qui calculent les fonctions suivantes :

$$\begin{array}{c} f(0,y) = \varphi_{C_0}(y) \\ f(1,y) = \varphi_{C_1}(y) \\ \vdots \\ f(2003,y) = \varphi_{C_{2003}}(y) \end{array}$$

où C_a est le programme numéro i avec l'entrée x fixée à $a : x = a$. Le théorème de paramétrisation dit que l'on peut calculer ces fonctions $\varphi_{C_a}(y)$ par une transformation de programme :

$$C_a = s(i,a)$$

où C_a est le programme pour calculer $\lambda y \,.\, \varphi_{C_a}(y)$ et a est le premier argument de la fonction $f(x,y)$ que l'on veut considérer comme paramètre, et la fonction s est récursive primitive.

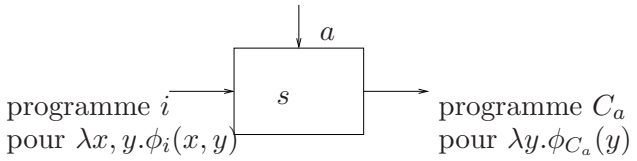


FIG. 1.1 – Illustration du théorème de paramétrisation.

Théorème 7 (s-m-n : cas scalaire) *Il existe une fonction s récursive primitive telle que pour tout x, y*

$$\varphi_i^{(2)}(x, y) = \varphi_{s(i,x)}^{(1)}(y) \tag{1.9}$$

La preuve du théorème (omise) peut être faite par une manipulation de machines de Turing.

Exemple. Soit g_a une fonction définie comme suit : $g_a = \lambda y \,.\, a + y^2$, nous voulons trouver le numéro de MT qui calcule g_a. Pour ceci, nous considérons a et y comme arguments, et la fonction $h(a, y) = a + y^2$ est calculable. Soit k le numéro de la machine de Turing qui calcule h. En utilisant le théorème **s-m-n** nous obtenons

$$h(a, y) = \varphi_k^{(2)}(a, y) = \varphi_{s(k,a)}^{(1)}(y) = a + y^2.$$

Théorème 8 (s-m-n : cas général) *Pour tout m et n il existe une fonction $s_n^m : \mathbb{N}^{m+1} \to \mathbb{N}$ récursive primitive telle que pour tout $\bar{x} \in \mathbb{N}^m$ et $\bar{y} \in \mathbb{N}^n$*

$$\varphi_i^{(m+n)}(\bar{x}, \bar{y}) = \varphi_{s_n^m(i,\bar{x})}^{(n)}(\bar{y}) \tag{1.10}$$

1.8.1 Exercices – s-m-n

1. Trouver une fonction h récursive primitive qui transforme un programme pour toute fonction récursive partielle $f(y)$ vers un programme pour la fonction $7f(y)$. C'est-à-dire $\varphi_{h(x)}(y) = 7\varphi_x(y)$.
2. Trouver une fonction g qui "compose" deux machines de Turing. C'est-à-dire g est telle que $\varphi_{g(p,q)}(x) = \varphi_p(\varphi_q(x))$.

Chapitre 2

Décidabilité et indécidabilité

2.1 Problèmes décidables

Comme nous avons déjà mentionné, les problèmes de décision que nous considérons sont posés sous la forme : "Étant donné x, est-ce que le prédicat $P(x)$ est vrai ?", ou bien sous la forme : "Est-ce que $x \in P$?" où P est un ensemble.

Nous somme maintenant capables de donner la définition centrale de ce cours : celle de la décidabilité :

Définition 15 *Un prédicat P est décidable (récursif) si χ_P est récursive totale. Au cas contraire on dit que P est indécidable.*

Une forme équivalente de cette définition est la suivante :

Définition 16 *P est récursif (décidable) s'il existe une fonction g récursive totale telle que :*

$$\forall x : g(x) = \begin{cases} 1 & si\ P(x), \\ 0 & si\ \neg P(x). \end{cases}$$

Intuitivement, P est décidable s'il existe un algorithme qui pour chaque x dit "OUI" ou "NON" à la question : "Est-ce que $P(x)$ est vrai ?".

Remarque. Par définition, la fonction caractéristique est définie partout, et si elle est calculable, alors elle est obligée d'être récursive totale.

Propriétés des prédicats décidables. *Fermeture booléenne* : Si A et B sont décidables, alors $\bar{A}$, $A \vee B$, $A \wedge B$ sont décidables. Pour prouver ce résultat, notons que

$$\begin{aligned} \chi_{\bar{A}}(x) &= 1 \dot{-} \chi_a(x) \\ \chi_{A \vee B}(x) &= sg(\chi_A(x) + \chi_B(x)) \\ \chi_{A \wedge B}(x) &= sg(\chi_{\bar{A}}(x) + \chi_{\bar{B}}(x)) \quad (\text{puisque } A \wedge B = \bar{A} \vee \bar{B}) \end{aligned}$$

2.2 Premiers problèmes indécidables

Dans la suite nous examinons deux prédicats représentants deux problèmes de décision importants. Nous avons maintenant tous les outils pour prouver leur indécidabilité.

1. Problème de l'arrêt [1] :

$$Arret(x, y) \triangleq \varphi_x(y) \downarrow \triangleq \text{ la machine de Turing } x \text{ s'arrête sur l'entrée } y$$

[1] "Halting problem" en anglais.

En langage courant ce problème est posé comme suit : est-ce que le programme x donne un résultat sur l'entrée y ?

2. Problème K : l'ensemble K est défini comme $K = \{x \mid \varphi_x(x) \downarrow\}$

$$K(x) \stackrel{\triangle}{=} \varphi_x(x) \downarrow \stackrel{\triangle}{=} \text{ le programme } x \text{ s'arrête sur l'entrée } x.$$

Le problème K est un sous-problème de l'arrêt (en fait $K(X) \Leftrightarrow Arret(x, x)$). Si l'on savait décider le problème de l'arrêt, on saurait aussi décider K.

Théorème 9 *K est indécidable.*

Démonstration :

Une fonction auxiliaire D'abord, observons que

$$v(x) = \begin{cases} 1 & \text{si } x = 0, \\ \uparrow & \text{si } x > 0. \end{cases}$$

est récursive partielle puisque $v(x) = 1 + \mu y\,.\,(x + y = 0)$.

Construction diagonale Nous allons prouver le théorème par contradiction. Supposons que K est décidable. Alors, il existe une fonction g récursive telle que :

$$g(x) = \begin{cases} 1 & \text{si } K(x), \\ 0 & \text{si } \neg K(x) \end{cases} = \begin{cases} 1 & \text{si } \varphi_x(x) \downarrow, \\ 0 & \text{si } \varphi_x(x) \uparrow. \end{cases}$$

Nous pouvons maintenant construire une nouvelle fonction :

$$d(x) = v(g(x)) = \begin{cases} 1 & \text{si } \varphi_x(x) \uparrow, \\ \uparrow & \text{si } \varphi_x(x) \downarrow. \end{cases}$$

La fonction d est récursive puisqu'elle est obtenue par la composition de v et de g qui sont récursives. Donc,

$$\begin{cases} d(x) \downarrow & \text{si } \varphi_x(x) \uparrow, \\ d(x) \uparrow & \text{si } \varphi_x(x) \downarrow. \end{cases}$$

ou, plus court, pour tout x nous avons

$$d(x) \downarrow \Leftrightarrow \varphi_x(x) \uparrow$$

Cette propriété nous permettra d'obtenir une contradiction.

Raisonnement "diagonale" Pour chaque a le fait que $d(a) \downarrow \Leftrightarrow \varphi_a(a) \uparrow$ implique que $d \neq \varphi_a$ (ces deux fonctions se comportent différemment sur a). Alors, d n'est pas dans la liste $\varphi_0, \varphi_1, \ldots, \varphi_a, \ldots$. Pourtant, cette liste est l'énumération de toutes les fonctions récursives partielles. Donc, d n'est pas récursive partielle, ce qui est une contradiction puisque nous avons montré précédemment que d est récursive partielle. On conclut que K est indécidable. ■

Vu l'importance du raisonnement ci-dessus nous le présentons encore une fois sous une forme différente.

Raisonnement "diagonale" - forme succincte De même, nous utilisons la fonction d définie dans la première preuve, qui est récursive partielle. Soit c son numéro. Alors, $d = \varphi_c$. Comme $d(x) \downarrow \Leftrightarrow \varphi_x(x) \uparrow$, nous avons

$$\forall x\ \varphi_c(x) \downarrow \Leftrightarrow \varphi_x(x) \uparrow .$$

En particulier, pour $x = c$ nous avons $\varphi_c(c) \downarrow \Leftrightarrow \varphi_c(c) \uparrow$, ce qui est une contradiction. Alors, K est indécidable. ■

Corollaire 1 *Le problème de l'arrêt est indécidable.*

Démonstration. Supposons que le problème de l'arrêt est décidable. Alors, il existe $f(x,y)$ récursive telle que

$$f(x,y) = \begin{cases} 1\downarrow & \text{si } Arret(x,y), \\ 0\uparrow & \text{sinon} \end{cases} = \begin{cases} 1\downarrow & \text{si } \varphi_x(y)\downarrow, \\ 0\uparrow & \text{si } \varphi_x(y)\uparrow. \end{cases}$$

Puis, la fonction $u(x) = f(x,x)$ est aussi récursive. En plus,

$$u(x) = \begin{cases} 1\downarrow & \text{si } \varphi_x(x)\downarrow, \\ 0\uparrow & \text{si } \varphi_x(x)\uparrow \end{cases} = \chi_K$$

Ceci veut dire que K est décidable. Mais, selon le théorème 9, K est indécidable, d'où une contradiction. Alors, le problème de l'arrêt est indécidable. ■

2.2.1 Exercices sur la diagonalisation

Cette série d'exercices sert à mieux comprendre la méthode de diagonalisation qui joue un rôle principal dans les preuves d'indécidabilité

1. **Le barbier** du village rase tous les habitants qui ne se rasent pas et seulement ceux-là. Est-ce qu'il se rase ? *Formaliser en utilisant le prédicat R(x,y). Comparer avec la preuve de l'indécidabilité du problème K. Réfléchir est-ce qu'il se rase vraiment.*
2. **Une fonction partielle qu'on ne peut pas compléter.** On considère la fonction suivante :

   $$f(x) = \varphi_x(x) + 1$$

   Montrer que :
   (a) f est récursive partielle.
   (b) il n'existe pas de fonction récursive totale g qui prolonge f, c'est-à-dire telle que

   $$g(x) = \begin{cases} f(x) & \text{si } f(x)\downarrow \\ \text{valeur arbitraire} & \text{si } f(x)\uparrow \end{cases}$$

   Indication : supposer le contraire et considérer le numéro de g
3. Montrer qu'il n'existe pas d'énumération effective de fonctions récursives totales, c'est à dire d'une liste $h_1(x), h_2(x), \ldots$ telle que :
   (a) chaque fonction $h(x)$ est récursive totale ;
   (b) chaque fonction récursive totale est dans la liste ;
   (c) la fonction $V(i,x) = h_i(x)$ est calculable.

   Pourquoi une telle énumération est quand même possible pour les fonctions récursives partielles ?

2.3 Méthode de réduction

Nous venons d'établir l'indécidabilité du problème de l'arrêt (corollaire 1) par le raisonnement suivant : le problème K se réduit au problème de l'arrêt, K est indécidable, donc $Arret$ est indécidable. Dans l'étude de complexité, il y a un raisonnement similaire : SAT $\propto$ Clique, SAT est NP-complet, alors Clique est NP-complet.

La *réduction* est une méthode très utile pour prouver l'indécidabilité de problèmes divers. Nous étudierons cette méthode de raisonnement plus profondément.

Définition 17 (m-réduction) *Soient A et B deux prédicats. $A \leq_m B$ s'il existe une fonction h récursive totale telle que $\forall x\ A(x) \Leftrightarrow B(h(x))$. Dans ce cas on dit que A se réduit à B par h*[2].

[2]Il existe d'autres notions de réduction utiles : 1-réduction, Turing-réduction, réduction polynômiale vue en AC, mais dans ces notes nous nous intéressons uniquement à la m-réduction.

Le diagramme suivant peut illustrer cette définition.

$$\begin{array}{ccccc} h(x) & \xrightarrow{B} & B(h(x)) & \text{oui/non} \\ \uparrow & & \downarrow Id & \\ x & \xrightarrow{A?} & A(x) & \text{oui/non} \end{array}$$

Proposition 8 (Propriétés de $\leq_m$)

- $A \leq_m A$ *(réflexivité)*
- $A \leq_m B \wedge B \leq_m C \wedge A \leq_m C$ *(transitivité)*
- *Si R est récursive et A est tel que $A \neq \emptyset$ et $\bar{A} \neq \emptyset$, alors $R \leq_m A$ (un problème décidable ce réduit à tous les problèmes)*

Le résultat suivant est presque évident : si A se réduit à B, et si l'on sait résoudre B, alors on peut résoudre A.

Lemme 5 *Si $A \leq_m B$ et B est décidable, alors A est décidable.*

Démonstration. Par définition de $\leq_m$, il existe une fonction h récursive totale telle que $\forall x A(x) \Leftrightarrow B(h(x))$. D'autre part, par définition de décidabilité, χ_B est récursive totale. Soit χ_A la fonction caractéristique de A. Donc,

$$\chi_A(x) = \begin{cases} 1 & \text{si } A(x) \\ 0 & \text{si } \neg A(x) \end{cases} = \begin{cases} 1 & \text{si } B(h(x)) \\ 0 & \text{si } \neg B(h(x)) \end{cases} = \chi_B(h(x)).$$

On peut voir que χ_A est récursive totale puisque elle est obtenue par la composition de χ_B et h qui sont récursives totales. Alors A est décidable. ■

Corollaire 2 *Si $A \leq_m B$ et A est indécidable, alors B est indécidable.*

Les résultats ci-dessus suggèrent une méthode pour prouver l'indécidabilité. Comme on sait que K est indécidable, afin de prouver que B est indécidable, il suffit de montrer que K se réduit à B (c-à-d $K \leq_m B$).

Cette méthode permet d'établir l'indécidabilité de nombreux problèmes. Nous donnerons un exemple simple et concret, et ensuite un résultat général.

Exemple. Nous considérons le problème de décision définie par $B(x) \equiv \varphi_x(3) = 10$. Il faut donc pour un x donné décider si la machine de Turing de numéro de Gödel x sur l'entrée 3 s'arrête et produit le résultat 10.

Nous montrons que B n'est pas récursif en utilisant une réduction de l'ensemble K à B. Soit g la fonction partielle définie comme suit

$$g(x,y) = \begin{cases} 10 & \text{si} \quad \varphi_x(x) \downarrow \\ \uparrow & \text{si} \quad \varphi_x(x) \uparrow \end{cases}$$

g est récursive partielle, comme $g(x,y) = U(x,x) \cdot 0 + 10$, où U dénote la fonction universelle (et cette fonction U est bien récursive partielle). Donc il existe un numéro n avec $g = \varphi_n$ et le théorème de "s-n-m" garantie que $g(x,y) = \varphi_{s(n,x)}(y)$. On a donc obtenu une fonction récursive totale $h(x) = s(n,x)$ telle que $g(x,y) = \varphi_{h(x)}(y)$.

Maintenant, pour montrer que $K(x) \Leftrightarrow B(h(x))$ nous considérons les deux cas possibles :

- Si $K(x)$, alors $\varphi_x(x) \downarrow$ et alors $\varphi_{h(x)}(y) = g(x,y) = 10$ pour tout y, et en particulier pour $y = 3$. Donc $\varphi_{h(x)}(3) = 10$ ce qui implique que $B(h(x))$.
- Si $\neg K(x)$, alors la fonction $\varphi_{h(x)}$ est définie nulle part et $\varphi_{h(x)}(3) \uparrow$. Donc $\neg B(h(x))$.

Nous venons de prouver que $K \leq_m B$ par h, d'où suit l'indécidabilité de B.

La même méthode permet d'établir un résultat d'indécidabilité très général

Théorème 10 (Théorème de Rice) *Soit $\mathcal{C}$ une classe non-triviale de fonctions récursives partielles. Alors, le prédicat $B_{\mathcal{C}}(x) \stackrel{\triangle}{=} \varphi_x \in \mathcal{C}$ est indécidable.*

"$\mathcal{C}$ non-trivial" veut dire que $\mathcal{C} \neq \emptyset$ et $\overline{\mathcal{C}} \neq \emptyset$. Par exemple, pour $\mathcal{C} = \{f \mid f(3) = 10\}$, nous avons $B_{\mathcal{C}}(x) \Leftrightarrow \varphi_x(3) = 10\}$.

Démonstration. Soit $\perp$ une fonction telle que $\perp(x) \uparrow$ pour tout x.

– Premier cas : $\perp \in \mathcal{C}$. Soit v une fonction récursive partielle telle que $v \notin \mathcal{C}$ (par l'hypothèse de non-trivialité elle existe), définissons :

$$g(x,y) = \begin{cases} v(y) & \text{si } \varphi_x(x) \downarrow, \\ \uparrow & \text{si } \varphi_x(x) \uparrow, \end{cases}$$

La fonction g peut être écrite comme $g(x,y) \stackrel{\triangle}{=} \varphi(x,x) \cdot 0 + v(y)$ et est donc récursive partielle, soit b son numéro. Par le théorème de paramétrisation $s-m-n$, $\varphi_b(x,y) = \varphi_{s(b,x)}(y)$. Soit $h(x) = s(b,x)$, qui est récursive totale.
Soit x une valeur arbitraire, alors soit $x \in K$ soit $x \notin K$. Pour le cas où $x \in K$, nous avons : $\varphi_{h(x)}(y) = v(y)$. En conséquence, $\varphi_{h(x)} = v$. En plus, par hypothèse, $v \notin \mathcal{C}$, donc $\varphi_{h(x)} \notin \mathcal{C}$. Nous en déduisons que $h(x) \notin B_{\mathcal{C}}$. Pour le cas où $x \notin K$, nous avons que

$$\varphi_{h(x)}(y) \uparrow \Rightarrow \varphi_{h(x)}(y) = \perp \Rightarrow \varphi_{h(x)} \in \mathcal{C}.$$

Donc $h(x) \in B_{\mathcal{C}}$. Ceci montre que $K \leq_m \overline{B_{\mathcal{C}}}$. Puisque K est indécidable, alors $\overline{B_{\mathcal{C}}}$ est indécidable et, par la fermeture booléenne, $B_{\mathcal{C}}$ l'est aussi. ◇

– Deuxième cas : $\perp \notin \mathcal{C}$. La preuve est similaire à celle du premier cas. Soit $w \in \mathcal{C}$ (elle existe par l'hypothèse de non-trivialité), définissons :

$$g(x,y) = \begin{cases} w(y) & \text{si } \varphi_x(x) \downarrow, \\ \uparrow & \text{si } \varphi_x(x) \uparrow, \end{cases}$$

Nous avons donc que la fonction $g(x,y) \stackrel{\triangle}{=} \varphi(x,x).0 + w(y)$ est récursive partielle, et soit a son numéro. Par le théorème $s-m-n$, nous avons $\varphi_b(x,y) = \varphi_{s(a,x)}(y)$. Soit $h(x) = s(a,x)$, qui est récursive totale.
Si $x \in K$ nous avons :

$$\varphi_x(x) \downarrow \Rightarrow g(x,y) = w(y) \Rightarrow \varphi_{h(x)}(y) = w(y).$$

Donc $\varphi_{h(x)} = w$ et $\varphi_{h(x)} \in \mathcal{C}$. Nous avons donc $h(x) \in B_{\mathcal{C}}$. Pour le cas où $x \notin K$ nous avons :

$$\varphi_x(y) \uparrow \Rightarrow \varphi_{h(x)}(y) \uparrow \Rightarrow \varphi_{h(x)} = \perp \Rightarrow \varphi_{h(x)} \notin \mathcal{C}.$$

Donc, $K \leq_m B_{\mathcal{C}}$ et $B_{\mathcal{C}}$ est indécidable. ■

Remarque. En termes usuels le théorème de Rice signifie que toutes les propriétés de fonctions calculables par MT sont indécidables.

Par exemple toutes les propriétés ci-dessous le sont

1. $\{x \mid Dom(\varphi_x)$ est infini$\}$
2. $\{x \mid \varphi_x$ est totale$\}$
3. $\{x \mid \varphi_x(20) \downarrow\}$
4. $\{x \mid \forall y : \varphi_x(y) = 3y^2 + 5y + 2\}$
5. $\{x \mid 3 \in Im(\varphi_x)\} = \{x \mid \exists y \ \varphi_x(y) = 3\}$

Les propriétés décidables des MT font toujours référence à la structure ou au fonctionnement des MT, et pas uniquement aux aspects fonctionnels, par exemple :

1. $\{x \mid \varphi_x(3)$ s'arrête avant 1000 pas$\}$
2. $\{x \mid$ la machine de Turing numéro x a un nombre pair d'instructions$\}$

2.4 Semi-décidabilité

Dans l'introduction nous avons discuté informellement les notions de semi-algorithme et d'un problème semi-décidable. Dans cette partie nous ferons une étude technique de ces concepts. Nous examinons d'abords quelques exemples.

Exemples

- $Arret(x, y) \stackrel{\triangle}{=} \varphi_x(y) \downarrow$. Nous avons prouvé que $Arret(x, y)$ est indécidable, mais il existe un semi-algorithme suivant : lancer la machine x sur l'entrée y, et si la machine s'arrête, renvoyer "OUI".
- $K(x) \stackrel{\triangle}{=} \varphi_x(x) \downarrow$. Pour ce problème indécidable il existe un semi-algorithme suivant : lancer la machine x sur l'entrée x, et si la machine s'arrête, renvoyer "OUI".
- $P(x) \stackrel{\triangle}{=} \varphi_x(3) = 10$. De même, nous avons prouvé que $P(x)$ est indécidable, et un semi-algorithme pour décider $P(x)$ est : lancer la machine x sur l'entrée 3 ; si la machine s'arrête et le résultat est 10, répondre "OUI".
- $Q(x) \stackrel{\triangle}{=} (Dom(\varphi_x)$ est infini). Il n'y a pas de semi-algorithme [3] et la raison intuitive est que l'on ne peut pas tester si une machine s'arrête sur un nombre infini d'entrées.

Définition 18 *L'ensemble P (comme avant $P \subseteq \mathbb{N}^k$) est* semi-décidable *s'il existe une fonction récursive partielle g telle que $P = Dom(g)$, c'est-à-dire $x \in P \Leftrightarrow g(x) \downarrow$. Autrement dit, g "s'arrête" sur tous les x dans P et "boucle" sur tous les $x \notin P$.*

Définition 19 *La fonction* semi-caractéristique *de P est :*

$$\psi_P(x) = \begin{cases} 1 & si\ x \in P, \\ \uparrow & si\ x \notin P. \end{cases}$$

Proposition 9 *Un prédicat P est semi-décidable si et seulement si sa fonction semi-caractéristique est récursive partielle.*

Démonstration. Pour $\Leftarrow$, supposons que ψ_P est récursive partielle. Alors, $P = Dom(\psi_P)$ et P est le domaine d'une fonction récursive partielle. Par définition, P est semi-décidable.

Pour $\Rightarrow$, supposons que P est semi-décidable. Donc, $P = Dom(g)$ où g est récursive partielle. Nous avons :

$$g(x) \downarrow \Leftrightarrow x \in P.$$

Soit

$$\psi_P(x) = \begin{cases} 1 & \text{si } x \in P, \\ \uparrow & \text{si } x \notin P. \end{cases}$$

$$\psi_P(x) = \begin{cases} 1 & \text{si } g(x) \downarrow, \\ \uparrow & \text{si } g(x) \uparrow. \end{cases}$$

Nous montrons que ψ_P est récursive partielle en écrivant $\psi_P(x) = g(x) \cdot 0 + 1$ ■

Proposition 10 *Si un prédicat P est décidable, alors il est semi-décidable.*

Un algorithme pour semi-décider P peut être comme suit : si $x \in P$ retourner "OUI", sinon boucler.

La démonstration ci-dessous formalise se raisonnement

Démonstration. Puisque P est décidable, alors sa fonction caractéristique

$$\chi_P(x) = \begin{cases} 1 & \text{si } x \in P \\ 0 & \text{si } x \notin P \end{cases}$$

[3] Prouver ceci est un exercice !

est récursive totale. La fonction semi-caractéristique de P est :

$$\psi_P(x) = \begin{cases} 1 & \text{si } \chi_P(x) = 1, \\ \uparrow & \text{si } \chi_P(x) = 0. \end{cases}$$

Nous allons maintenant prouver que ψ_P est récursive partielle. En fait, on peut écrire ψ_P sous forme : $\psi_P(x) = w(\chi(x))$ où la fonction

$$w(y) = \begin{cases} 1 & \text{si } y = 1, \\ \uparrow & \text{si } y = 0. \end{cases}$$

peut être exprimée comme : $w(y) = \mu z.(z \cdot y = 1)$. Alors, v est récursive partielle, et en plus χ_P est récursive totale, nous en déduisons que ψ_P l'est aussi.

■

Forme normale

Proposition 11 *Chaque ensemble semi-décidable P peut-être représenté comme suit :*

$$P(x) \Leftrightarrow \exists t\ R(x,t)$$

où R est un prédicat récursif primitif.

L'idée derrière cette proposition est que $R(t,x)$ vérifie est-ce que le semi-algorithme pour P sur l'entrée x s'arrête après t pas avec le résultat "OUI".
Démonstration. D'après le théorème de la forme normale pour les fonctions récursives partielles, la fonction $g(x)$ telle que $P = Dom(g)$ peut être représentée comme $g(x) = h(x, \mu\, t.R(x,t))$ où h et R sont récursives primitives. Alors,

$$x \in P \Leftrightarrow g(x)\downarrow \Leftrightarrow h(x, \mu\, t.R(x,t))\downarrow \Leftrightarrow \mu t.R(x,t)\downarrow \Leftrightarrow \exists t\ R(x,t).$$

En effet, nous venons de montrer que $x \in P \Leftrightarrow \exists t\ R(x,t)$. ■

2.5 Ensembles récursivement énumérables (r.e.)

Dans cette section nous donnons une caractérisation complètement différente de la semi-décidabilité.

Définition 20 *P est récursivement énumérable si et seulement s'il existe une fonction récursive totale h telle que $P = Im(h)$ ou bien si $P = \emptyset$, c'est-à-dire, $P = \{h(0), h(1), \ldots, h(i), \ldots\}$ (ou P est vide). On dit que h énumère P.*

Intuitivement, P est récursivement énumérable si et seulement s'il y a un algorithme qui imprime la liste de tous les éléments de l'ensemble.

Interprétation
- Semi-décidabilité = acceptation par une MT.
- Récursivement énumérable = génération par une MT.

Dans d'autres cours vous avez vu plusieurs résultats qui relient l'acceptation avec la génération, par exemple

Acceptation par machine à pile ⇔ Génération par grammaire hors contexte

Nous allons prouver un résultat important du même type pour les MT.

Théorème 11 *Semi-décidable ⇔ récursivement énumérable.*

Démonstration. ($\Leftarrow$) Soit P un prédicat récursivement énumérable. On veut montrer que P est semi-décidable.

Voici un semi-algorithme pour semi-décider si $x \in P$.

$i = 0$
while $(h(i) \neq x)$ {
  $i++$
}
return "OUI"

Il faut noter que h est totale, c-à-d elle est définie partout. Donc, le test $h(i) \neq x$ donne toujours un résultat.

Formellement, si $P = \emptyset$, il est clair qu'il est semi-décidable. Si $P \neq \emptyset$, alors $P = Im(h) = \{h(0), h(1), \ldots, h(i), \ldots\}$ où h est récursive totale. Puis, nous définissons une fonction $f(x) \stackrel{\triangle}{=} \mu i.h(i) = x$. Nous avons

$$x \in P \Leftrightarrow \exists i \; h(i) = x \Leftrightarrow \mu \, i.h(i) = x \downarrow \Leftrightarrow f(x) \downarrow .$$

Donc, $P = Dom(f)$. Comme f est récursive partielle, P est semi-décidable.

($\Rightarrow$)Maintenant, supposons que P est semi-décidable, alors $P = Dom(g)$ où g est récursive partielle. On suppose que P est non-vide (l'autre cas est trivial) et on cherche à énumérer P. Par le théorème de la forme normale,

$$P = \{x \mid \exists t R(x,t)\}$$

où R est récursive primitive. Essentiellement, nous allons générer P pour chaque (x,t) comme suit : si $R(x,t)$ alors on peut générer x, sinon on génère a où a est un élément fixe de P. Plus concrètement, soit $A = 2^x \cdot 3^t$, définissons une fonction h pour énumérer P comme suit :

$$h(A) = \begin{cases} ex2(A) & \text{si } R(ex2(A), ex3(A)), \\ a & \text{sinon.} \end{cases}$$

Un programme pour calculer $h(A)$ est le suivant.

fonction $h(A)$
  $x \leftarrow ex2(A)$
  $t \leftarrow ex3(A)$
  if ($h(x)$ converge en t pas) **then return** x
  else return a

Il suffit ensuite de prouver que $P = Im(h)$ (notons que h est récursive totale). Pour cela, nous allons prouver que $x \in P \Leftrightarrow x \in Im(h)$.

Effectivement, soit $x \in P$. Alors $\exists t \; R(x,t)$. Puis, pour $A = 2^x \cdot 3^t$ nous avons $h(A) = x$, et donc $x \in Im(h)$.

Soit maintenant x un élément de $Im(h)$. Alors $x = h(A)$ pour un certain A. Nous avons ensuite deux cas : $x = a$ et $x \neq a$. Si $x = a$, il est clair que $x \in P$. Si $x \neq a$, alors pour $t = ex3(A)$, nous avons que $R(x,t)$ est vrai. Autrement dit, $\exists t \; R(x,t)$, ce qui implique que $x \in P$. ■

Le résultat suivant relie la décidabilité avec la semi-décidabilité (et explique le dernier terme).

Théorème 12 (Théorème de Post) *Un prédicat P est décidable ssi P est semi-décidable et $\overline{P}$ est semi-décidable.*

L'intuition derrière ce théorème est que si l'on sait dire "OUI" quand x est dans P aussi bien que dire "NON" quand x n'est pas dans P, alors on sait décider si $x \in P$.

Démonstration. Supposons que P est décidable. Nous allons prouver que P et $\overline{P}$ sont semi-décidables. Le fait que P est décidable implique (en vigueur de la Proposition 10) que P est semi-décidable. D'autre part, si P est décidable, alors $\overline{P}$ est décidable (par fermeture) et donc $\overline{P}$ est semi-décidable.

Inversement, étant donné que P est semi-décidable et $\overline{P}$ est semi-décidable, il faut prouver que P est décidable. Comme P et $\overline{P}$ sont récursivement énumérables, nous avons $P = Im(f)$ et $\overline{P} = Im(h)$ avec f et h récursives totales, c'est-à-dire,

$$P = \{f(0), f(1), \ldots\}$$
$$\overline{P} = \{h(0), h(1), \ldots\}$$

Pour décider si $x \in P$, on peut utiliser l'algorithme suivant (qui converge toujours) :

```
repeat {
  i ← 0
  if f(i) = x then return "OUI"
  if h(i) = x then return "NON"
  i++
}
```

Plus formellement, nous définissons

$$u(x) \stackrel{\triangle}{=} \mu i.(f(i) = x \ \vee \ h(i) = x)$$

Il est clair que u est récursive totale puisque soit $x \in P = Im(f)$ soit $x \in \overline{P} = Im(g)$. La fonction

$$\chi_P(x) = \begin{cases} 1 & \text{si } f(u(x)) = x, \\ 0 & \text{sinon.} \end{cases}$$

est récursive totale, et P est donc décidable. ■

Théorème 13 ***Propriétés de fermeture des ensembles récursivement énumérables***

1. *Si les prédicats P et Q sont récursivement énumérables, alors les prédicats $P \wedge Q$ et $P \vee Q$ sont récursivement énumérables.*
2. *Pour certains prédicats récursivement énumérable P leur négation $\neg P$ n'est pas récursivement énumérable.*
3. *Si $P(x, y)$ est r.e., alors le prédicat (avec quantification existentielle non-bornée) $\exists y \ P(x, y)$ est aussi récursivement énumérable.*

Démonstration.

1. $(\wedge)$ Supposons que P et Q sont récursivement énumérables. Donc ils sont semi-décidables, et il existe deux fonctions récursives partielles f et g telles que $P(x) \Leftrightarrow f(x) \downarrow$ et $Q(x) \Leftrightarrow g(x) \downarrow$, d'où nous déduisons que $P(x) \wedge Q(x) \Leftrightarrow (f(x) + g(x)) \downarrow$. Nous avons établi que $P \wedge Q$ est le domaine d'une fonction récursive partielle $f + g$, donc il est semi-décidable. ◇

   $(\vee)$ Supposons que P et Q sont récursivement énumérables et non-vides. Donc $P = Im(v)$ et $Q = Im(w)$ pour deux fonctions récursives totales v et w. La façon la plus simple pour énumérer $P \vee Q$ est la suivante :

   $$v(0), w(0), v(1), w(1), v(2), w(2), \ldots$$

   Formellement, $P \vee Q = Im(h)$ avec

   $$h(n) = \begin{cases} v(n \, div \, 2) & \text{si} \quad n \text{ est pair} \\ w(n \, div \, 2) & \text{si} \quad n \text{ est impair} \end{cases}$$

   ◇

2. Le prédicat K est semi-décidable parce que $K(x) \Leftrightarrow \varphi_x(x) \downarrow \Leftrightarrow U(x, x) \downarrow$ et donc $K = Dom(f)$ avec $f(x) = U(x, x)$ - une fonction récursive partielle. Si son complément $\overline{K}$ était aussi semi-décidable, alors K serait décidable par théorème de Post. Or, c'est impossible. Donc K est semi-décidable, et son complément non. ◇

3. Supposons que $P(x,y)$ est semi-décidable et que $Q(x) \equiv \exists y\ P(x,y)$. Par théorème de forme normale $P(x,y) \Leftrightarrow \exists t\ R(x,y,t)$ (avec R récursif primitif), d'où

   $$Q(x) \Leftrightarrow \exists y \exists t\ R(x,y,t) \Leftrightarrow \exists A\ R(x, ex2(A), ex3(A)) \Leftrightarrow (\mu A.R(x, ex2(A), ex3(A)))\downarrow \Leftrightarrow f(x)\downarrow$$

   avec $f(x) = \mu A.R(x, ex2(A), ex3(A))$ récursive partielle. Nous avons représenté Q comme domaine d'une fonction récursive partielle f, par conséquent Q est semi-décidable. ■

Dans la pratique la preuve de semi-décidabilité peut se faire en utilisant les propriétés de fermeture ci-dessus et la semi-décidabilité de certains prédicats.

Proposition 12 (Quelques prédicats r.e. utiles) *Les prédicats suivants sont r.e. :*
- *tous les prédicats décidables ;*
- *le prédicat Resultat défini comme suit :*

  $$Resultat(x,y,z) \Leftrightarrow (\varphi_x(y) = z)$$

- *Arret ;*

Démonstration : Tous les prédicats décidables sont r.e. selon la proposition 10.
La fonction semi-caractéristique de *Resultat* peut s'écrire comme

$$\psi_{Resultat}(x,y,z) = \psi_{=}(U(x,y), z),$$

donc elle est récursive partielle, d'où suit la semi-décidabilité de *Resultat*.
Finalement $Arret(x,y) \Leftrightarrow U(x,y)\downarrow$, donc $Arret = Dom(U)$ est aussi semi-décidable. ■

Exemple Nous utiliserons les deux dernières propositions pour prouver la semi-décidabilité du prédicat suivant

$$S(x) \equiv Dom(\varphi_x) \text{ contient au moins deux nombre premiers}$$

Effectivement, s admet la représentation suivante :

$$S(x) \equiv \exists y_1 \exists y_2 \Big(EstPremier(y_1) \wedge EstPremier(y_2) \wedge y_1 \neq y_2 \wedge Arret(x,y_1) \wedge Arret(y_2) \Big)$$

qui utilise uniquement des prédicats semi-décidables et des opérations préservant la semi-décidabilité. ◇

2.6 Semi-décidabilité vs. réduction

Cette section n'a pas été présentée dans les cours de cette année 2003.

Proposition 13 *Si $A \leq_m B$ et B est semi-décidable (r.e.), alors A est semi-décidable.*

Démonstration. Supposons que $A \leq_m B$ pour h récursive totale, c'est-à-dire

$$x \in A \Leftrightarrow h(x) \in B. \tag{2.1}$$

D'autre part, comme B est semi-décidable, $B = Dom(f)$ où f est récursive partielle. Alors, en combinant avec (2.1), nous avons

$$x \in A \Leftrightarrow h(x) \in B \Leftrightarrow h(x) \in Dom(f) \Leftrightarrow f(h(x))\downarrow$$

En conséquence, $A = Dom(f \circ h)$ et A est alors semi-décidable. ■

Corollaire 3 *Si $A \leq_m B$ et B est non-semi-décidable, alors A est non-semi-décidable.*

Théorème 14 *Soit $T = \{x \mid \varphi_x \text{ est totale}\}$ [4]. Alors,*

1. *T n'est pas récursivement énumérable.*
2. *T n'est pas co-récursivement énumérable (c'est-à-dire $\overline{T}$ n'est pas r.e.)*

Démonstration. Nous allons d'abord prouver (1) par contradiction. Supposons que T est récursivement énumérable. Alors,

$$T = g(0), g(1), g(2), \ldots$$

où g est récursive totale. Puis,

$$\varphi_{g(0)}, \varphi_{g(1)}, \varphi_{g(2)}, \cdots \tag{2.2}$$

est la liste de toutes les fonctions récursives totales. Soit

$$\begin{aligned} d(x) &= \varphi_{g(x)}(x) + 1 \\ &= U(g(x), x) + 1. \end{aligned} \tag{2.3}$$

Notons que $d(x)$ est récursive partielle. Puisque $\varphi_{g(x)}$ est totale, alors $\forall x\ \varphi_{g(x)} \downarrow$ et nous avons donc $\forall x\ d(x) \downarrow$. Autrement dit, $d(x)$ est en effet récursive totale et W est dans la liste (2.2), c-à-d $\exists c\ d = \varphi_{g(c)}$. D'où, $d(c) = \varphi_{g(c)}(c)$. D'autre part, par définition (2.3), $d(c) = \varphi_{g(c)}(c) + 1$. Alors, $d(c) = d(c) + 1$, une contradiction. Par conséquent, T n'est pas récursivement énumérable. ■

Pour prouver (2), c'est-à-dire $\overline{T}$ n'est pas récursivement énumérable, il suffit de faire la réduction $\overline{K} \leq_m \overline{T}$ (voir les exercices ci-dessous).

2.7 Exercices – analyse de décidabilité de problèmes

Faire l'analyse de décidabilité des prédicats P suivants :

1. $P(x) \equiv \forall y.\varphi_x(y) = y^2$; déduire qu'il y a une infinité de MT qui calculent la fonction y^2
2. $P(x) \equiv 10 \in \mathsf{Im}(\varphi_x)$;
3. $P(x) \equiv \mathsf{Dom}(\varphi_x)$ est infinie ;
4. $P(x) \equiv \varphi_x$ est totale ;
5. $P(x) \equiv \varphi_x(3) \downarrow$;
6. $P(x) \equiv \varphi_x(x) = x$ (Examen 2000) ;
7. $P(x) \equiv \overline{K}(x) \equiv \varphi_x(x) \uparrow$;

Plan de l'analyse :

1. *Définir P en français.*
2. *Est-ce que P est récursivement énumérable (r.e.) ?*
3. *Est-ce que P est décidable (récursif) ?*
4. *(Question subsidiaire) est-ce que $\overline{P}$ est r.e. ?*

[4] T correspond à l'ensemble de tous les programmes qui ne bouclent pas.

Chapitre 3

Applications de la calculabilité dans l'informatique

3.1 Problèmes de vérification

Nous nous intéressons au problème très pratique de vérification de programmes, qui a souvent la forme suivant : étant donné un programme P (avec un modèle de calcul MC) et une propriété Q (spécification), est-ce que P satisfait Q ?

Nous allons dans la suite étudier la décidabilité et l'indécidabilité de quelques problèmes de ce type. Il est important de noter que l'on peut parler de la décidabilité ou l'indécidabilité d'une classe de problèmes et pas seulement d'une instance de problème. Nous allons considérer de différents modèles de calculs (avec mémoire finie et infinie, sans ou avec entrées).

3.1.1 Machines de Turing

Les machines de Turing sont un modèle de calcul avec mémoire infinie. Soit M la machine de Turing (qu'on peut représenter par son numéro de Gödel x) que l'on veut vérifier. Nous mentionnons dans la suite quelques problèmes décidables et indécidables (certains résultats ci-dessous sont triviaux, certains ont été établis dans le Chapitre 2, les autres peuvent être établis facilement en utilisant les mêmes techniques).

- Problèmes décidables (et même récursifs primitifs) :
  - Si Q est une propriété syntaxique (par exemple, "est-ce que la machine M a plus que 2002 états", ou moins que 2003 instructions, ou aucune instruction du type "aller à gauche ?", etc.)
  - Si Q est une propriété de complexité bornée (par exemple, "est-ce que la machine M s'arrête sur l'entrée y avant t pas avec le résultat z ?").
- Problèmes indécidables :
  - Est-ce que la machine M s'arrête sur y ? (ce problème est r.e.).
  - Est-ce que la machine M s'arrête au moins sur une entrée ? (r.e.).
  - Est-ce que la machine M s'arrête sur chaque entrée ? (ce problème n'est pas r.e.).
  - Est-ce que la machine M calcule la fonction $f(y) = y$? (ce problème n'est pas r.e. non plus).
  - ...

3.1.2 Automate fini

Un automate fini est un modèle de calcul avec mémoire finie, qui peut être vu comme un programme sans variables. *Grosso modo* tous les problèmes concernant les automates finis sont décidables, par exemple :

- L'automate A a 156 états.
- A accepte y (y appartient au langage régulier accepté par A).
- $L(A) \neq \emptyset$ (A accepte au moins une entrée).
- $L(A) = \Sigma^*$ (A accepte toutes les entrées).
- $L(A) = (a+b)^*bba$.
- $L(A) = L(B)$.

Les algorithme de décision pour ces problèmes ont été vus dans le cours d'Automates et Langages.

3.1.3 Machine à pile

Machine à pile est un modèle de calcul avec mémoire infinie et une entrée. Rappelons qu'elle est composée d'un automate fini et une pile. Les opérations sur une telle machine sont : empiler, dépiler, sommet de la pile, etc. Un programme sur une machine à pile peut avoir des instructions de la forme "**if** .. **then** .. **else** ", par exemple :

q_0 : empiler(0) ; **goto** q_1
q_1 : empiler(1) ; **goto** q_2
q_2 : dépiler ; **goto** q_3
q_3 : **if** (sommet=1) **goto** q_4 **else goto** q_0
q_4 : lire(a) ; **if** $(a = 1)$ **goto** q_4 **else goto** q_0

Exemples des problèmes décidables et indécidables concernant les machines à pile sont :

- Problèmes décidables (algorithmes vus en cours d'Analyse Syntaxique) :
  - Machine à pile A a 156 états.
  - A accepte y (y est dans le langage hors-contexte).
  - $L(A) \neq \emptyset$.
- Problèmes indécidables (indécidabilité admise sans preuve) :
  - $L(A) = \Sigma^*$.
  - $L(A) \cap L(B) \neq \emptyset$.

3.1.4 Machine à deux piles

Dans cette sections nous démontrons l'indécidabilité du problème d'arrêt pour les machines à 2 piles sans entrée. Ce résultat lui-même n'est pas très intéressant mais il permet d'illustrer une technique de preuve d'indécidabilité extrêmement utile : celle de la *réduction par simulation*. Pour simplifier la simulation nous supposons que le sommet de la pile vide est 0, et qu'on peut dépiler de la pile vide (elle reste vide).

Théorème 15 *Le problème de l'arrêt est indécidable pour les machines à 2 piles.*

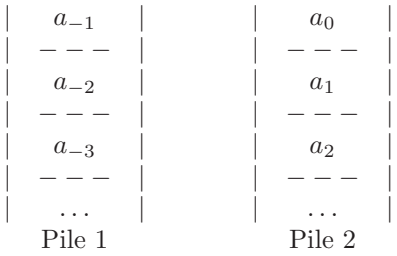
Démonstration. L'idée essentielle est de construire pour chaque machine de Turing T une machine à 2 piles $B = B(T)$ qui simule T, c'est-à-dire B a le même comportement que T. En conséquence, T s'arrête si et seulement si $B(T)$ s'arrête. Nous allons prouver que *Arret* (pour la machine de Turing) $\leq_m$ *Arret* (pour la machine à 2 piles) et donc le problème de l'arrêt pour les machines à 2 piles est indécidable.

La méthode de réduction par simulation a 3 étapes :

Représenter chaque configuration de T par une configuration de B : la configuration suivante de T

$$\begin{array}{ccccccccccc} \ldots & | & a_{-2} & | & a_{-1} & | & a_0 & | & a_1 & | & a_2 & | & \ldots \\ & & & & & & \triangle & & & & & & \\ & & & & & & q & & & & & & \end{array}$$

est représentée par l'état q de la machine B et par les piles suivantes :



Programmer (sur la machine B) chaque instruction de T :

Instructions de T	*Instructions de B*
$qc \to qc'$	q : **if** $sommet(P_2) = c$ **goto** q_1 q_1 : dépiler(P_2) ; **goto** q_2 q_2 : empiler(P_2, c') ; **goto** q'
$qc \to q'c'D$	q : **if** $sommet(P_2) = c$ **goto** q_1 q_1 : dépiler(P_2) ; **goto** q_2 q_2 : empiler(P_1, c') ; **goto** q'
$qc \to q'c'G$	Exercice

En appliquant cette traduction à chaque instruction de la machine de Turing T on obtient un programme $B(T)$ pour une machine à 2 piles. Ce programme fait la même chose que T en utilisant une autre structure de données (2 piles au lieu d'un ruban)

Analyser l'initialisation/arrêt : On voit facilement que la machine de Turing T s'arrête sur l'entrée x si et seulement si la machine à 2 piles s'arrête en démarrant de $P_1 = \epsilon$ et $P_2 = 01^x$. On a donc réduit le problème de l'arrêt pour les MT au problème de l'arrêt pour les machines à 2 piles. Comme le premier problème est indécidable, le second l'est également. ■

Une conséquence intéressante de cette démonstration est que les machines à 2 piles peuvent calculer toutes les fonctions récursives partielles.

3.1.5 Exercice – Machine à compteurs

Nous représentons l'étude de ce modèle de calcul sous la forme d'une série d'exercices

Définition informelle. Une machine à n compteurs (MàCn) a plusieurs registres x_1, x_2,..., x_n chacun capable de stocker un nombre naturel.

un programme - c'est un ensemble d'instructions de types suivants :

1. $q : x++$; **goto** p
2. $q : x--$; **goto** p
3. q :**if** $x = 0$ **then goto** p **else goto** r
4. q : **stop**

La tentative de calculer $0--$ produit une erreur. La configuration de la MàCn est le $n+1$-uplet $(q, x_1, x_2, \ldots, x_n)$ qui représente l'étiquette de l'instruction courante et le contenu de tous les compteurs. On suppose que pour l'entrée x la machine démarre dans la configuration $(\text{init}, x, 0, \ldots, 0)$ où "init" est une instruction particulière, et qu'elle calcule jusqu'à un **stop** . Le résultat de ce calcul est le contenu du registre x_1 après l'arrêt.

Le but de ce TD consiste à prouver l'indécidabilité du problème d'arrêt pour les machines à 4 compteurs (et équivalence des MàC4 aux MT)

1. Programmer une machine à 2 compteurs (MàC2) qui calcule $f(x) = x \bmod 2$.
2. Simuler une pile (pour simplicité sur $\{0, 1\}$) par 2 compteurs :
   – Comment représenter le contenu de pile $a_0a_1a_2\ldots$ par des valeurs de x_1 et x_2 ?

- Quelle transformation de x_1, x_2 correspond a l'opération Empiler(0) ? Programmer cette transformation sur la MàC.
- Même question pour Empiler(1)
- Même question pour Dépiler ;
- Même question pour **if** Sommet=0 **then goto** p **else goto** r

3. Simuler 2 piles sur 4 compteurs
4. Prouver qu'une MàC4 peut calculer chaque fonction récursive partielle
5. Prouver que le problème d'arrêt est indécidable pour les MàC4
6. Prouver que le problème d'arrêt est indécidable pour les MàC2
7. Prouver que le problème d'arrêt est décidable pour les MàC1

Nous retenons le résultat principal

Théorème 16 *Les machines à 2 compteurs peuvent simuler les machines de Turing. Alors, le problème de l'arrêt est indécidable pour les machines à 2 compteurs.*

3.1.6 Problèmes pratiques

En pratique, on s'intéresse à vérifier des programmes C (ou Pascal, ou Lustre). Par exemple nous voulons savoir si un programme s'arrête sur une entrée x, ou s'il s'arrête toujours, ou s'il calcule $f(x) = x!$. La décidabilité dépend en général du modèle de calcul.

- (a) Pour le cas d'un ordinateur avec mémoire finie (bornée), on peut le considérer comme un automate fini (dont le nombre d'états est souvent très grand) et en théorie tous les problèmes de vérification d'automates finis sont décidables. Pourtant la taille de l'automate peut rendre l'application des méthodes pour les automates finis aux programmes infaisable en pratique.
- (b) Si l'on permet des variables non-bornées, alors le modèle de l'ordinateur peut simuler une machine à compteurs, et tous les problèmes qui sont décidables pour le cas (a) deviennent indécidables.
- (c) Si le modèle de calcul a un nombre non-borné de variables (registres, cellules de mémoire), alors tous les problèmes précédents sont aussi indécidables.

En général, la plupart de problèmes de vérification de programmes sont indécidables. Parmi les directions de recherche actuellement poursuivies mentionnons :

- Chercher de bonnes classes de programmes pour lesquels le problème de vérification est décidable.
- Trouver de méthodes de "Bonne programmation", qui consiste à accompagner chaque programme par une preuve de correction.
- Utiliser des semi-algorithmes.

3.2 Problèmes indécidables dans la théorie de langages

3.2.1 Problèmes relatives aux langages hors contexte

Ici on énumère sans preuve quelques résultats concernant les grammaires hors contexte. Ce n'est pas par hasard que ces résultats ressemblent à ceux de la section 3.1.3.

On utilise les notations G et H pour des grammaires hors contexte(sur un alphabet Σ), $L(G)$ et $L(H)$ pour leurs langages engendrés, w pour un mot sur Σ.

- Problèmes décidables :
  - Problème de mot : est-ce que $w \in L(G)$?
  - Problème de langage vide : est-ce que $L(G) = \emptyset$?
- Problèmes indécidables (indécidabilité admise sans preuve) :
  - Problème de langage universel : est-ce que $L(G) = \Sigma^*$?
  - $L(G) \cap L(H) = \emptyset$?
  - Problème d'équivalence : est-ce que $L(G) = L(H)$?

3.2.2 Systèmes de réécriture

Dans cette section nous étudions un type de grammaires très général.

Soit Σ un alphabet fini. Un système de réécriture sur l'alphabet Σ est un ensemble fini Π de règles de la forme $v \to w$ (où v et w sont des mots sur l'alphabet Σ). On dit qu'un mot B est dérivable de A s'il peut être obtenu en appliquant plusieurs fois les règles de Π à des sous-mots (notation : $A \xrightarrow{*} B$).

Le but de cette section consiste à étudier la décidabilité du problème $\mathsf{Dérivable}(\Pi, A, B)$: *"Est-ce que* $A \xrightarrow{*} B$ *dans* Π*"* .

Par exemple A et B sont des génomes de deux cellules, les règles $v \to w$ représentent des mutations possibles, et on veut décider si B peut être un descendant de A

Exemple : deux instances du problème. Soit $R1 = ab \to ba$ et $R2 = a \to aa$.

- Est-ce que $aab \xrightarrow{*} aaaba$? La dérivation $a\underline{ab} \xrightarrow{R1} \underline{a}ba \xrightarrow{R2} \underline{a}aba \xrightarrow{R2} aaaba$ montre que $aaaba$ est dérivable de aab.
- Est-ce que $aba \xrightarrow{*} ababa$? On peut remarquer, que l'application de règles $R1$ et $R2$ préserve le nombre de b dans le mot. Comme aba contient un seul b, tandis que $ababa$ en contient deux, la dérivation est impossible.

Simulation. On veut prouver l'indécidabilité du problème $\mathsf{Dérivable}(\Pi, A, B)$ en y réduisant le problème de l'arrêt pour les machines à 2 compteurs.

Pour une Mà2C M on construira un système de réécriture $\Pi(M)$ qui simule cette machine. On représentera la configuration $c = (q, x, y)$ par le mot $\widehat{c} = \$1^x q 1^y \$$.

On représentera dans une table tous les types d'instructions de la Mà2C, comment ces instructions modifient la configuration c, comment doit ce transformer le mot $\widehat{c}$ et quelles sont les règles qui assurent une telle transformation

in	$c \to c'$	$\widehat{c} \to \widehat{c'}$	$\widehat{in}$
q : $x{+}{+}$; **goto** p	$(q,x,y) \to (p,x+1,y)$	$\$1^x q 1^y \$ \to \$1^{x+1} p 1^y \$$	$q \to 1p$
q : $x{-}{-}$; **goto** p	$(q,x,y) \to (p,x-1,y)$	$\$1^x q 1^y \$ \to \$1^{x-1} p 1^y \$$	$1q \to p$
q :**if** $x=0$ **then** p	$(q,0,y) \to (p,0,y)$	$\$q1^y\$ \to \$p1^y\$$	$\$q \to \$p$
else r	$(q,x+1,y) \to (r,x+1,y)$	$\$1^{x+1} q 1^y \$ \to \$1^{x+1} r 1^y \$$	$1q \to 1r$
q : **stop**	$(q,x,y) \to$	$\$1^x q 1^y \$ \xrightarrow{*} \varepsilon$	$1q \to q,\ q1 \to q,$ $\$q\$ \to \varepsilon$

(les règles concernant y sont symétriques à celles pour x et on s'est permis de les omettre).

L'algorithme de construction du système de réécriture $\Pi(M)$ est donc le suivant : pour chaque instruction in de la machine M prendre la règle (ou les règles) de $\widehat{in}$ (quatrième colonne), et les mettre tous ensemble. Ça donne un ensemble fini de règles de réécriture.

On va maintenant prouver que le système de réécriture $\Pi(M)$ simule la machine M.

Soit $c = (q, x, y)$ une configuration de M et $\widehat{c} = \$1^x q 1^y \$$ le mot qui la représente. Soit in l'instruction étiquetée par q dans M.

Lemme 6 (simulation d'un pas de calcul) *Si c n'est pas une configuration de l'arrêt, alors $c \xrightarrow{in} c'$ dans M si et seulement si $\widehat{c} \xrightarrow{\widehat{in}} \widehat{c'}$ dans $\Pi(M)$.*

Démonstration :

On voit tout de suite de la table ci-dessus, que si $in \neq$ **stop** , alors une seule règle de $\Pi(M)$ s'applique à $\widehat{c}$: si $in = {+}{+}$, c'est la seule règle de $\widehat{in}$. Si $in =$ **if** - c'est la "bonne" règle des deux de $\widehat{in}$. Dans tous les cas on voit que cette règle fait la même transformation que in. □

Lemme 7 (simulation de stop) *Si c est une configuration de l'arrêt , alors $\widehat{c} \xrightarrow{*} \varepsilon$ dans $\Pi(M)$.*

Démonstration : Si c est une configuration de l'arrêt, c-à-d $in =$ **stop** , alors on peut appliquer les règles de $\widehat{in}$ pour supprimer d'abord tous les 1, et à la fin supprimer le $\$q\$$. □

Lemme 8 (correction) *M possède un calcul qui mène à l'arrêt à partir d'une configuration c si et seulement si $\widehat{c} \xrightarrow{*} \varepsilon$ dans $\Pi(M)$*

Démonstration : Supposons que M s'arrête à partir de la configuration c. Soit c' la dernière configuration de ce calcul (c'est forcément une configuration de l'arrêt). Alors en appliquant le lemme 7 pour chaque pas de calcul on obtient une dérivation $\widehat{c} \xrightarrow{*} \widehat{c'}$ dans $\Pi(M)$. En appliquant la propriété B pour la configuration de l'arrêt c', on obtient que $\widehat{c'} \xrightarrow{*} \varepsilon$. En concaténant ces deux dérivations ensemble on obtient que $\widehat{c} \xrightarrow{*} \varepsilon$.

Supposons maintenant que M ne s'arrête pas à partir de la configuration c. Dans ce cas-là le calcul de M est soit infini, soit mène à une erreur $(0--)$, mais jamais dans un état de l'arrêt. La propriété A nous garantit que les réécritures de $\Pi(M)$ à partir du mot $\widehat{c}$ ne mènent jamais à un mot qui contient l'étiquette q' d'une instruction **stop** . Donc on ne pourra jamais utiliser la règle $\$q'\$ \to \varepsilon$ - la seule qui permet d'obtenir le mot vide. On a prouvé donc que $\widehat{c} \not\xrightarrow{*} \varepsilon$. □

En particulier, Arrêt2C(M,x) si et seulement si $\$1^x\mathsf{init}\$ \xrightarrow{*} \varepsilon$, c'est à dire, si et seulement si Dérivable$(\Pi(M), \$1^x\mathsf{init}\$, \varepsilon)$. Donc on a réduit Arrêt2C à Dérivable (on a montré que Arrêt2C$\leq_m$ Dérivable). Comme Arrêt2C est indécidable, Dérivable est aussi indécidable. ■

Bibliographie

[1] Pierre Wolper. *Introduction à la calculabilité : Cours et exercices corrigés*, 2e édition, Dunod, 2001,

[2] H. Rogers. Theory of Recursive Functions and Effective Computability, MIT Press, 1987.

[3] Ch. Boitet. Récursivité, calculabilité, application à la théorie des langages : notes de cours - Grenoble : Université scientifique et médicale, 1983,1987...