CALCULABILITE ET COMPLEXITE

cours M2 du certificat C3 de la Maîtrise

S. Cerrito, H. Comon, C. Crépeau

1993-1994

1 Introduction

Au début du vingtième sciècle, les mathématiciens se sont posés un problème fondamental: donner des bases sûres à leur discipline. La question apparaissait de façon cruciale avec la mise en évidence de paradoxes comme le paradoxe de Russel (l'ensemble des ensembles qui ne se contiennent pas) ou, plus simplement, le paradoxe du menteur: "cette phrase est un mensonge".

Pour résoudre ces problèmes, il était nécessaire de préciser la notion de vérité mathématique. David Hilbert proposa alors d'identifier la vérité d'un ensemble d'énoncés avec leur cohérence. Mais que signifie "cohérence"? Cette notion ne peut avoir de sens que pour des énoncés décrits dans un langage suffisamment formel. On dira alors qu'un ensemble d'énoncés est cohérent si l'on ne peut pas prouver à partir de cet ensemble un énoncé et sa négation . Mais qu'est-ce qu'une preuve? Les preuves doivent être "mécanisables". Plus précisément, étant donné un ensemble fini de règles de construction, on doit pouvoir vérifier mécaniquement qu'une suite d'énoncés est bien construite suivant ces règles (on dit alors que c'est une preuve de l'énoncé final). Mais reste le problème de savoir ce que signifie "mécaniquement".

La première idée qui vient à l'esprit est d'utiliser un procédé physique. Mais un tel procédé (une "machine") ne peut non plus être satisfaisant. Par exemple, on peut imaginer construire une machine qui lance des dés et qui détermine la valeur d'une fonction $f(n)$ par le $n$ième lancé de dés. On ne peut prétendre alors que la machine *calcule* la fonction f car chaque utilisation de la machine conduira à une fonction différente. On ne peut pas non plus imaginer une telle machine vérifiant qu'une séquence d'énoncés est une preuve. En fait, on a besoin de machines dont les calculs sont *reproductibles*. De façon générale, les machines concrètes sont exclues car elles ne peuvent être totalement fiables (que se passe-t-il lorsque le serveur tombe en panne? En déduit-on qu'on ne peut calculer la fonction en cours d'évaluation?). y C'est ainsi que de nombreux mathématiciens travaillèrent à la définition de *modèles de calcul*.

K. Gödel utilise en 1930 la notion de *fonction récursive* comme modèle de calcul, ce qui lui permet d'établir peu après son fameux théorème d'incomplétude, répondant ainsi par la négative à une autre question fameuse de Hilbert. La théorie des fonctions récursives a été par la suite développée par de nombreux mathématiciens comme Ackermann et Kleene. Les fonctions récursives sont introduites extrèmement succintement au chapitre 2.

A. Church, vers 1930 également, se penche sur un tout autre modèle: le λ-calcul. À l'origine, celui-ci était conçu comme une alternative à la théorie des ensembles. Mais, s'il n'a pas eu de succès dans cette perspective, il s'est avéré par contre bien adapté à la description des fonctions (calculables). Le λ-calcul sert d'ailleurs de modèle de calcul aux langages de programmation fonctionnels. Ce modèle fait l'objet du chapitre 4.

Malheureusement, le λ-calcul pur ne permet pas de résoudre les paradoxes, parce qu'il n'y a en λ-calcul, aucune différence entre les fonctions et les objets auxquels elles s'appliquent. Par exemple, on peut parfaitement appliquer une fonction à elle-même. Church avait parfaitement mesuré les défauts du système qu'il avait proposé et il introduisit lui-même dans les années 40 le λ-calcul *typé*. Outre l'intérêt mathématique et logique de ce nouveau formalisme (aspects sur lesquels nous ne nous étendons pas), il correspond aussi à des préoccupations informatiques fondamentales, comme l'analyse statique des programmes. C'est ainsi qu'il est à la base des modèles utilisés pour des langages fonctionnels typés comme ML. Nous verrons le λ-calcul typé au chapitre 6.

Enfin, bien d'autres modèles de calcul ont été introduits pour diverses applications (on peut citer, entre autres, Kolmogorov, Markov, Minski,...). Il est hors de question de les passer en revue. Cependant, il en est un qui a particulièrement interessé (et continue d'intéresser) les informaticiens. Il s'agit d'un modèle proposé par Turing et par Post en 1936. L'histoire n'a retenu que le nom de Turing pour appeler ces machines "machines de Turing", parce que Turing publia son article quelques mois avant Post. Turing est ainsi souvent considéré comme le père de l'informatique. La raison du grand succès de ce modèle est l'avènement des calculateurs, sur le modèle de la machine (cette fois-ci bien concrète) de Von Neumann. En effet, la machine de Von Neumann peut-être vue comme la réalisation matérielle du système inventé par Turing. (On ne sait pas d'ailleurs dans quelle mesure Von Neumann a pompé sur Turing). Nous parlerons des machines de Turing aux chapitres 3, 7 et 8.

En fait tous les modèles de calculs introduits (sauf le λ-calcul typé) sont équivalents, c'est-à dire qu'ils définissent, sur les entiers, la même notion de fonction calculable (ce que nous prouvons dans le chapitre 5). Ces résultats d'équivalence viennent à l'appui d'une affirmation de Church connue sous le nom de *thèse de Church*: "les fonctions calculables sur les entiers sont les fonctions définissables en λ-calcul". Cependant, certains modèles sont mieux adaptés que d'autres pour l'expression ou la résolution de problèmes. Il en est d'ailleurs de même des langages de programmation (et ce n'est pas un hasard).

La calculabilité (l'étude des modèles de calcul) pouvait donc être considére jusque vers 1950 comme une branche des mathématiques ou de la logique. Mais il se trouve que les modèles définis par les mathématiciens permettent justement de répondre à des questions fondamentales de l'informatique. En effet, la notion de calcul ainsi modélisée, il est possible de comparer des calculs. Il est aussi possible d'établir les limites théoriques de ce que peut faire une machine. Il existe en effet des fonctions qu'il n'est pas possible de programmer. Le langage de programmation utilisé n'a ici pas d'importance, puisque les modèles de calcul sous-jacents sont tous équivalents. Voici un exemple, appelé "problème de l'arrêt", décrit dans le langage PASCAL:

Exemple 1.1 *Peut-on écrire dans le langage PASCAL une fonction* $Arret(F, D{:}Text){:}Boolean$ *qui décide si une fonction donnée en PASCAL dans le fichier F, s'arrêtera ou bouclera à l'infini sur une donnée D? Plus exactement, la question est de savoir s'il est possible d'écrire une fonction PASCAL*

$$Arret(F, D{:}Text){:}Boolean$$

avec les propriétés suivantes :

- *l'argument F de type Text contient le code d'une fonction PASCAL FU à un argument de type Text laquelle donne une valeur booleenne*
- *l'argument D contient une donnée pour FU*
- *Arret renvoie True si FU ne boucle pas a l'infini sur la donnée D et Faux sinon*

On montre que ce *n'est pas possible d'écrire* une telle fonction PASCAL, ni d'ailleurs aucune fonction analogue dans un autre langage. Nous verrons que la *fonction d'arrêt* n'est pas calculable.

Une fois définie de façon rigoureuse la notion de fonction calculable, on pourra définir de façon rigoureuse la notion de **problème décidable** (= problème pour lequel il existe une fonction calculable dont le résultat est oui ou non) et étudier, pour des problèmes spécifiques, s'il sont décidables ou pas. Par exemple, nous verrons que le problème de l'arrêt est indécidable.

Les modèles de calcul permettent aussi de comparer et d'évaluer précisément des algorithmes (ce qui ne serait pas possible autrement sans dépendre d'aspects matériels). En effet, si les algorithmes sont décrits par des programmes d'un même modèle de calcul, on peut évaluer et comparer leur efficacité: on appellera *complexité en temps* d'un algorithme le nombre d'étapes de calcul nécéssaire dans le modèle considéré. De même, en évaluant la taille mémoire nécéssaire dans le modèle considéré, on obtient une notion de *complexité en espace*. Ces notions sont introduites dans le chapitre 8.

De plus, même si une fonction est calculable, son calcul peut requérir un temps (ou un espace) beaucoup trop important pour son calcul effectif. Nous verrons (dans le modèle des machines de Turing) qu'il existe des problèmes qui sont décidables mais pour lesquels il n'existe aucun algorithme efficace permettant de les résoudre. On peut ainsi construire une hiérarchie de complexité des problèmes (sans réference explicite à un algorithme). Ces aspect seront étudiés dans le chapitre 9.

PARTIE I :CALCULABILITE

2 Fonctions récursives

On commence par considérer une formalisation de la notion intuitive d'algorithme basée sur le concept mathématique de *fonction récursive*.

Considérons les fonctions partielles à valeur entière de plusieurs arguments:

$$\bigcup_{k \in N} \{f : N^k \to N\}$$

avec la notation $f(\vec{n}) \uparrow$ pour indiquer que f n'est pas définit au point $\vec{n} = n_1, n_2, ..., n_k$. Nous allons extraire de cet ensemble deux sous-ensembles que l'on nomme les *fonctions récursives* et les *fonctions semi-récursives*. Le premier ensemble ne contient que des fonctions *totales*, c'est à dire définies partout. Il est défini par induction à partir de l'ensemble suivant de fonctions initiales.

Définition 2.1 *Les* <u>*fonctions initiales*</u> *sont Pi_i^k, S et Z où*

$$Pi_i^k(n_1, n_2, ..., n_k) = n_i \textit{ pour } 1 \leq i \leq k$$

$$S(n) = n + 1, Z(n) = 0$$

On définit d'abord les opérations de composition, récursion primitive et minimisation sur un ensemble de fonctions totales afin de définir ensuite l'ensemble des fonctions (totales) récursives. On dit d'un ensemble F de fonctions totales que

Définition 2.2 *F est fermé par* <u>*composition*</u> *si $\forall \chi, \psi_1, \psi_2, ..., \psi_m \in F$, la fonction φ définie par*

$$\varphi(\vec{n}) = \chi(\psi_1(\vec{n}), \psi_2(\vec{n}), ..., \psi_m(\vec{n}))$$

est dans F.

Définition 2.3 *F est fermé par* <u>*récursion primitive*</u> *si $\forall \chi, \psi \in F$, la fonction φ définie par*

$$\varphi(m, \vec{n}) = \begin{cases} \chi(\vec{n}) & si\ m = 0 \\ \psi(\varphi(m-1, \vec{n}), m-1, \vec{n}) & si\ m > 0 \end{cases}$$

est aussi dans F.

Définition 2.4 *F est fermé par* <u>*minimisation*</u> *si* $\forall \chi \in F$ *telle que* $\forall \vec{n} \exists m \ [\chi(\vec{n}, m) = 0]$ *la fonction* φ *définie par*

$$\varphi(\vec{n}) = \min_m [\chi(\vec{n}, m) = 0]$$

est dans F.

Définition 2.5 *l'ensemble* $\Re$ *des* <u>*fonctions (totales) récursives*</u> *est le plus petit ensemble contenant les fonctions initiales qui soit fermé par composition, récursion primitive et minimisation.*

Autrement dit, les fonctions récursives sont toutes les fonctions que l'on peut construire à partir des fonctions initiales par composition, récursion primitive et minimisation. Toutes les fonctions qui nous sont familières, telles que $a + b, a - b, a \times b, a \, mod \, b, a \, div \, b, a^b, ...$, sont des fonctions récursives. Ce formalisme a fortement inspiré la description de plusieurs langages de programmation tels que LISP ou SCHEME où l'on a un ensemble de fonctions initiales (équivalentes à celles données précédemment) et des règles de constructions par composition et récursion.

Exemple 2.1 $f(a, b) = a + b$ *est une fonction récursive. Elle est définie par*

$$f(a,b) = \begin{cases} \chi(b) & si \ a = 0 \\ \psi(f(a-1,b), a-1, b) & si \ a > 0 \end{cases}$$

avec $\chi(x) = x = Pi_1^1(x)$ *et* $\psi(x, y, z) = x + 1 = S(Pi_1^3(x, y, z))$*. Puisque* Pi_1^1, Pi_1^3 *et S sont initiales donc récursives,* ψ *est récursive par composition et f par récursion primitive.*

2.1 Fonctions semi-récursives

L'ensemble des fonctions *semi-récursives*, que l'on va maintenant définir, contient toutes les fonction récursives (totales) mais peut aussi contenir des fonctions qui ne sont pas définies pour certains arguments

Dans le cas de la composition des fonctions partielles, on prend la convention que $\chi(\psi_1(\vec{n}), \psi_2(\vec{n}), ..., \psi_m(\vec{n})) \uparrow$. si $\psi_i(\vec{n}) \uparrow$.

Définition 2.6 *F est fermé par* <u>*minimisation-partielle*</u> *si pour une fonction totale $\chi \in F$*

$$\varphi(\vec{n}) = \min_m[\chi(\vec{n}, m) = 0] \in F$$

avec la convention que $\varphi(\vec{n}) \uparrow$ si $\forall m \ [\chi(\vec{n}, m) > 0]$.

Définition 2.7 *l'ensemble $\wp\Re$ des* <u>*fonctions semi-récursives*</u> *est le plus petit ensemble contenant les fonctions récursives qui soit fermé par composition et minimisation-partielle.*

Autrement dit, les fonctions semi-récursives sont toutes les fonctions que l'on peut construire à partir des fonctions récursives par composition et minimisation-partielle.

Exemple 2.2 $f(a, b) \begin{cases} = 0 & si\ a = b = 0 \\ \uparrow & sinon \end{cases}$
est une fonction semi-récursive. Elle est définie par

$$f(a, b) = \min_m[a + m = 0] + \min_m[b + m = 0]$$

Puisque $a + b$ est récursive (Exemple 2.1) alors $\min_m[x + m = 0]$ est semi-récursive par minimisation-partielle et de même f est semi-récursive par composition.

Sur cet exemple, on peut remarquer que, quoique la fonction f ne soit pas définie partout, il existe une fonction récursive qui dit si oui ou non un couple d'entiers est dans le domaine de définition de f. On peut donc "compléter" la définition de f en une définition de fonction récursive. Ce n'est pas toujours le cas.

La notion de fonction récursive constitue une formalisation de la notion intuitive de fonction calculable. L' équation :

$$\text{Fonction récursives} = \text{Fonctions calculables}$$

est un analogue de la *thèse de Church.*

Maintenant, passons plutôt à la formalisation correspondante de la notion d'ensemble décidable. Informellement, on dit qu'un ensemble est décidable s'il existe un algorithme qui est capable de décider si un objet donné appartient ou pas à l'ensemble.

2.2 Ensembles récursifs et récursivement énumérables

La notion d'ensemble récursif que l'on va définir est une formalisation de la notion d'ensemble décidable (d'entiers non-négatifs).

De même, la notion d'ensemble récursivement énumerable est une formalisation de la notion d'ensemble semi-décidable. On rappelle que l'on dit qu'un ensemble E est semi-décidable quand il existe un algorithme qui, pour tout élement $a \in E$, "prouve" que $a \in E$. Un ensemble peut être semi-décidable sans être décidable : voir l'exemple de l'ensemble des tautologies du calcul des prédicats.

Soit $L \subseteq N$, un ensemble d'entiers non-négatifs. La fonction *caractéristique* γ_L de L est définie par

$$\gamma_L(n) = \begin{cases} 1 & \text{si } n \in L \\ 0 & \text{si } n \notin L \end{cases}$$

et la fonction *partielle caractéristique* φ_L de L est définie par

$$\varphi_L(n) \begin{cases} = 1 & \text{si } n \in L \\ \uparrow & \text{si } n \notin L \end{cases}$$

Définition 2.8 *L est un <u>ensemble récursif</u> si γ_L est une fonction récursive.*

Exemple 2.3 *$L = \{n \mid n$ est premier$\}$ est récursif.*

Exemple 2.4 *Quand il est formalisé correctement*
$L = \{n \mid n$ interpreté en ASCII est une fonction PASCAL syntaxiquement correcte $\}$ *est récursif.*

Définition 2.9 *L est un <u>ensemble récursivement énumérable</u> (noté <u>r.é.</u>) si φ_L est une fonction semi-récursive.*

Exemple 2.5 *$L = \{n \mid \exists p, q$ premiers $, 2n = p + q\}$ est r.é mais peut-être pas récursif (La conjecture de Goldbach stipule que L est l'ensemble de tous les entiers.)*

Exemple 2.6 *Quand il est formalisé correctement*
$L = \{n \mid$ la $n^{ième}$ fonction PASCAL s'arrêtera sur donnée $n\}$ est r.é. mais non récursif.

Les équations :

$$\text{Récursif} = \text{Décidable}$$

$$\text{Recursivement Enumerable} = \text{Semi-décidable}$$

(analogues à la thèse de Church) sont justifiées (pas formellement prouvées!) par plusieurs résultats de logique mathématique. La théorie des fonctions récursives a été elaborée à partir des travaux de Gödel et Kleene des années 30.

2.3 Exercices

2.3.1

Parmi les 3 fonctions suivantes, 2 sont récursives, et pour la dernière, on ne sait pas actuellement si elle est récursive ou non. Pouvez-vous dire (en le justifiant) quelles sont les deux fonctions dont on sait qu'elles sont récursives?

$$f_1(x) = \begin{cases} 1 & \text{si Dieu existe} \\ 0 & \text{sinon} \end{cases}$$

$$f_2(x) = \begin{cases} 1 & \begin{array}{l}\text{si le développement décimal de } \pi \text{ contient} \\ \text{au moins } x \text{ 5 consécutifs}\end{array} \\ 0 & \text{sinon} \end{cases}$$

$$f_3(x) = \begin{cases} 1 & \begin{array}{l}\text{si le développement décimal de } \pi \text{ contient} \\ \text{exactement } x \text{ 5 consécutifs}\end{array} \\ 0 & \text{sinon} \end{cases}$$

2.3.2

On dit qu'une fonction récursive est récursive primitive si on peut la définir sans utiliser l'opération de minimisation, mais en utilisant le cas échéant la constante 0. Montrer que les fonctions suivantes sont récursives primitives :
a) La fonction produit: $\times(x, y)$.
b) Toute fonction constante, à un nombre quelconque d'arguments.
c) La fonction $\overline{sg}(x)$ ou : $\overline{sg}(x) = 1$ si $x = 0$, $\overline{sg}(x) = 0$ sinon.

d) La fonction $sg(x)$ ou : $sg(x) = 0$ si $x = 0, sg(x) = 1$ sinon.
e) La fonction $Exp(x, y)$ ou $Exp(x, y) = x^y$ avec la convention $Exp(0, 0) = 1$.
f) La fonction prédécesseur : $pred(x) = 0$ si $x = 0$, $pred(x) = x - 1$ sinon.
g) La fonction soustraction d'entiers naturels, qui vaut $x - y$ si $y \leq x$ et 0 sinon.
h) La fonction factorielle : x!.

2.3.3

Montrer que l'ensemble des relations récursives primitives est fermé par rapport aux opérations booléennes.

2.3.4

Montrer que, si R est une relation récursive primitive et f, g sont deux fonctions récursives primitives, alors la fonction

$$h(\vec{x}) = \text{ if } R(\vec{x}) \text{ then } f(\vec{x}) \text{ else } g(\vec{x})$$

est elle aussi récursive primitive. Étendre ces résultats aux définitions par cas.

2.3.5

Soient R une relation récursive primitive et f une fonction récursive primitive. Montrer que la fonction g définie par

$$g(\vec{x}) = \bigwedge_{0 \leq i \leq f(\vec{x})} R(\vec{x}, i)$$

est une relation récursive primitive.

2.3.6

On dit qu'un ensemble F de fonctions est fermé par *maximisation bornée* si, pour toute fonction χ, ψ de F telles que

$$\forall \vec{n} \exists m \leq \psi(\vec{n}).(\chi(\vec{n}, m) = 0)$$

la fonction φ définie par :

$$\varphi(\vec{n}) = \max_{m \leq \psi(\vec{n})} [\chi(\vec{n}, m) = 0]$$

appartient à F. Montrer que l'ensemble des fonctions récursives primitives est fermé par maximisation bornée.

De même, dit qu'un ensemble F de fonctions est fermé par *minimisation bornée* si, pour toute fonctions χ, ψ de F telles que

$$\forall \vec{n} \exists m \leq \psi(\vec{n}).(\chi(\vec{n}, m) = 0)$$

la fonction φ définie par :

$$\varphi(\vec{n}) = \min_{m \leq \psi(\vec{n})} [\chi(\vec{n}, m) = 0]$$

appartient à F.

Montrer que l'ensemble des fonctions récursives primitives est fermè par minimisation bornée.

2.3.7

Démontrer le resultat suivant :

> Soit f une fonction primitive récursive à un argument, strictement croissante et valant 0 en 0. Alors la fonction g définie par :
> $$y = g(x) \text{ si et seulement si } f(y) \leq x < f(y+1)$$
> est récursive primitive.

2.3.8

Soit le résultat suivant, dû à Kleene:

Théorème
Il existe une fonction récursive à deux arguments $En(x, y)$ qui énumere toutes les fonctions récursives primitives à un argument. C'est-à-dire : $En(x, y) = z$ si et seulement si la x-ème fonction primitive récursive a comme résultat z quand son argument est y.

Utiliser le théorème ci-dessus pour démontrer que l'ensemble des fonctions primitives récursives est strictement inclus dans l'ensemble des fonctions récursives.

2.3.9

Montrer que les fonctions suivantes sont semi-récursives.
Sont elles récursives primitives ? Sont elles récursives?
a) La fonction quotient euclidien : $quot(x,y) = [\frac{x}{y}]$ et la fonction reste.
b) La fonction $div(x,y)$ ou $div(x,y) = 1$ si y est multiple de x, $div(x,y) = 0$ sinon.

2.3.10

Soit $PASC_k$ l'ensemble des fonctions PASCAL ayant k paramètres entier en entrée et retournant une valeur entière. Soit $PASC_k^i$ l'ensemble des fonctions de $PASC_k$ contenant i caractères. Considérons la fonction $SUP(n)$ qui à chaque entier n associe la valeur maximale que peut écrire une fonction de $PASC_0^n$ (il n'y a qu'un nombre fini de fonctions dans $PASC_0^n$ donc l'une d'elle retourne une valeur plus grande que toutes les autres). C'est-à-dire

$$SUP(n) = \max_{f \in PASC_0^n} \{val(f)\}$$

Montrer que la fonction $SUP(n)$ n'est pas "calculable en PASCAL", c'est-à-dire, qu'il n'existe pas de fonction PASCAL S dont, pour tout n, l'exécution termine et donne pour résultat $SUP(n)$.

2.3.11

Utiliser les définitions du cours et les exercices précédents pour montrer que l'on a les inclusions strictes suivantes:
fonc. rec.prim. $\subset$ fonc. rec. $\subset$ fonc. semi-rec. $\subset$ fonc. sur les nombres naturels.

2.3.12 Complément de cours

La fonction d'Ackermann est définie de la façon suivante :

$$\begin{array}{ll} A(n,m) = m+1 & \text{si } n = 0 \\ A(n,m) = A(n-1,1) & \text{si } n > 0, m = 0 \\ A(n,m) = A(n-1, A(n,m-1)) & \text{si } n > 0, m > 0 \end{array}$$

Montrer que la fonction d'Ackermann n'est pas récursive primitive mais est récursive.

3 Les Machines de Turing

Bien que la théorie des fonctions récursives nous fournisse une formalisation juste de la notion de calculabilité, nous restons neanmoins insatisfaits, car les notions de machine et de langage de programmation n'interviennent même pas dans cette approche. La machine de Turing est une modélisation mathématique très simple, mais très puissante, du fonctionnement d'un ordinateur. Ce modèle développé par le mathématicien Alan Turing (en 1936) ressemble essentiellement à ce que l'on voit sur la figure 3.

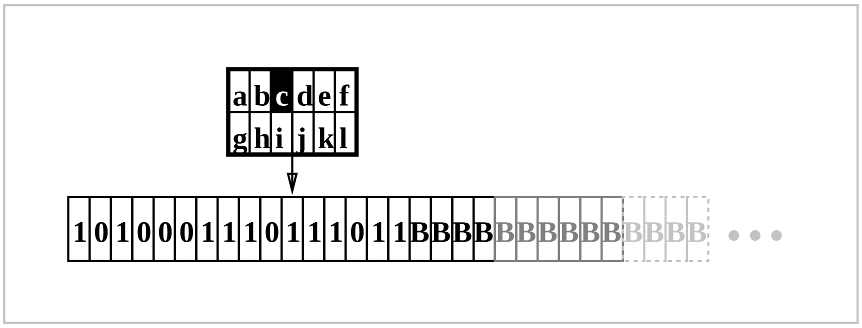

Figure 1: Machine de Turing

La machine travaille sur un ruban borné à gauche, mais infini à droite, où chaque case peut contenir un symbole d'un alphabet fini donné. La machine en elle même est un dispositif pouvant se trouver dans un nombre fini d'états. Elle possède une table lui permettant de changer d'état en fonction de son état actuel et du symbole qu'elle voit sur son ruban de travail. À chaque étape de son calcul, elle peut changer d'état, remplacer le symbole qu'elle voyait sur le ruban par un autre et ensuite se déplacer d'une case à gauche ou à droite.

Définition 3.1 *Une* machine de Turing *est un septuplet*

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

où

- $Q = \{q_0, q_1, ..., q_k\}$ *est l'ensemble des* états *de la machine,*

- $\Gamma = \{c_0, c_1, ..., c_n\}$ *est l'alphabet de travail, (pour simplifier on suppose* $Q \cap \Gamma = \emptyset$*),*
- $B \in \Gamma$ *est un symbole spécial associé à une case vide (toutes les cases sauf un nombre fini contiennent* B*),*
- $\Sigma \subseteq \Gamma - \{B\}$ *est l'ensemble des symboles avec lesquels les mots fournis en entrée à la machine sont exprimés,*
- q_0 *est l'*état initial*),*
- $F \subseteq Q$ *est l'ensemble des* états acceptants *et*
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\lhd, \bigtriangledown, \rhd\}$ *est la* fonction de transition.

Définition 3.2 *Une* description instantanée *(DI), aussi appellée* configuration *d'une machine* M *est un mot* $\alpha_1 q \alpha_2$ *avec* $q \in Q$ *et* $\alpha_1, \alpha_2 \in \Gamma^*$*. Une configuration de la forme* $q_0\alpha$ *où* q_0 *est l'état initial et* $\alpha \in \Sigma^*$ *est dite* initiale.

Une machine M est dans la configuration $\alpha_1 q \alpha_2$ si elle est dans l'état q et sa tête de lecture pointe sur le premier symbole de α_2 (si α_2 est vide alors le symbole courant est B). Le mot α_1 contient tous les symboles à gauche de la tête de lecture. Tous les symboles du ruban à droite de α_2 sont nécessairement des Bs.

La fonction de transition δ permet à la machine de transformer son état, de modifier le symbol qu'elle voit et de se déplacer vers la gauche ($\lhd$), vers la droite ($\rhd$) ou rester sur place ($\bigtriangledown$) en fonction de son état courant et du symbol qu'elle voit présentement. Cette fonction n'est pas nécessairement définie pour tous les points possibles de $Q \times \Gamma$.

Définition 3.3 *Etant donnée une machine* M *la relation de* transition $\vdash_M$ *sur les configurations de* M *est définie par*

- $\alpha x_1 q x_2 \beta \vdash_M \alpha q' x_1 y \beta$ *si* $\delta(q, x_2) = (q', y, \lhd)$.
- $\alpha q x \beta \vdash_M \alpha y q' \beta$ *si* $\delta(q, x) = (q', y, \rhd)$.
- $\alpha q x \beta \vdash_M \alpha q' y \beta$ *si* $\delta(q, x) = (q', y, \bigtriangledown)$.

Quand on a $\alpha_1 q \alpha_2 \vdash_M \alpha_3 q' \alpha_4 \vdash_M \ldots \vdash_M \alpha_i p \alpha_{i+1}$ *on écrit*

$$\alpha_1 q \alpha_2 \vdash_M^* \alpha_i p \alpha_{i+1}$$

et on dit que $\alpha_i q \alpha_{i+1}$ *découle de* $\alpha_1 q \alpha_2$.

Observons que si $\delta(q, x) = (y, \lhd)$, alors il n'existe pas de transition à partir de la configuration $qx\alpha$ (la tête de lecture ne peut pas quitter le ruban...).

Définition 3.4 *Etant donnée une machine M, un* calcul *de M sur la donnée α est une suite de configurations* $q_0\alpha \vdash_M c_1 \vdash_M \ldots c_n \vdash_M \ldots$.

Le calcul d'une Machine de Turing se déroule à partir de la configuration initiale, où la machine se trouve dans l'état q_0 avec sa tête de lecture sur la première case du ruban de travail. Un mot $\alpha \in \Sigma^*$ est placé sur le ruban de la machine à partir du début du ruban et tout le reste du ruban contient des Bs. Tant que la fonction δ sera définie sur l'état courant et le symbole sur lequel la tête de lecture pointe couramment, la machine va utiliser δ pour passer d'une configuration à sa suivante jusqu'à ce qu'elle aboutisse sur une paire (q, σ) où δ n'est pas définie. La machine s'arrête alors.

Définition 3.5 *Le langage accepté par une machine de Turing M est:*

$$L(M) = \{\omega \mid \omega \in \Sigma^* \text{ et } q_0\omega \vdash_M^* \alpha_1 q \alpha_2 \text{ pour un } q \in F \text{ et des } \alpha_1, \alpha_2 \in \Gamma^*\}$$

Par convention, on écrit toujours δ de façon à ce qu'elle ne soit jamais définie pour les élément de F afin de forcer M à s'arrêter aussitôt qu'une configuration acceptante a été atteinte. Quand M est bien définie par le contexte on remplace $\vdash_M$ et $\vdash_M^*$ par $\vdash$ et $\vdash^*$.

Exemple 3.1 *Comme premier exemple de description de machine de Turing, considérons la machine suivante qui accepte le langage* $\{0^n 1^n \mid n \geq 0\}$.

$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, X, Y, B\}, \delta, q_0, B, \{q_4\})$$

où δ est donnée sous forme de la table donnée en figure 2.

La machine fonctionne en remplaçant successivement un 0 *par* X *et un* 1 *par* Y *jusqu'à l'épuisement de l'un ou l'autre. Si les* 0 *et les* 1 *sont épuisés en même temps alors M accepte sinon elle rejette le mot fourni en entrée. Un exemple du calcul de M sur la donnée* $\omega = 0011$ *est fourni dans la figure 3.*

	$Q \times \Gamma$	0	1	X	Y	B
initialisation	q_0	$(q_1, B, \triangleright)$	-	-	-	$(q_4, B, \triangleright)$
recherche des 1	q_1	$(q_1, 0, \triangleright)$	$(q_2, Y, \triangleleft)$	$(q_1, X, \triangleright)$	$(q_1, Y, \triangleright)$	-
recherche des 0	q_2	$(q_1, X, \triangleright)$	$(q_2, 1, \triangleleft)$	$(q_2, X, \triangleleft)$	$(q_2, Y, \triangleleft)$	$(q_3, B, \triangleright)$
1 épuisé ?	q_3	-	-	$(q_3, X, \triangleright)$	$(q_3, Y, \triangleright)$	$(q_4, B, \triangleright)$
état final	q_4	-	-	-	-	-

Figure 2: la fonction de transition δ

$q_0 0011$	$\vdash Bq_1 011$	$\vdash B0q_1 11$	$\vdash Bq_2 0Y1$	
	$\vdash BXq_1 Y1$	$\vdash BXYq_1 1$	$\vdash BXq_2 YY$	
	$\vdash Bq_2 XYY$	$\vdash q_2 BXYY$	$\vdash Bq_3 XYY$	
	$\vdash BXq_3 YY$	$\vdash BXYq_3 Y$	$\vdash BXYYq_3$	$\vdash BXYYBq_4$

Figure 3: le calcul de $M(0011)$

3.1 langages et fonctions Turing-calculables

Définition 3.6 *Un langage $L \subseteq \Sigma^*$ est* Turing-semi-calculable *s'il existe une machine de Turing M telle que $L = L(M)$. L est* Turing-calculable *si de plus M s'arrête sur tous les mots $\omega \in \Sigma^*$.*

La notion de langage Turing-calculable est une autre formalisation de la notion d'ensemble décidable et, comme on verra, est équivalente à celle d'ensemble récursif. De même, la notion de langage Turing-semi-calculable est une autre formalisation de la notion d'ensemble semi-décidable, équivalente à celle d'ensemble r.é.

En plus de pouvoir accepter des langages, les machines de Turing peuvent être utilisées pour calculer des fonctions (éventuellement pas totales) à valeurs entières de plusieurs arguments. L'approche classique à de tels calculs est d'associer à l'entier n le mot 0^n et de même d'associer au vecteur $\vec{n} = n_1, n_2, ..., n_k$ le mot $0^{n_1}10^{n_2}1...10^{n_k}$. Si une machine M lors de son calcul sur mot $0^{n_1}10^{n_2}1...10^{n_k}$ s'arrête en laissant 0^m sur son ruban, on dit alors que $M(\vec{n}) = m$. Il se peut que pour certains $\vec{n}$ la machine M ne s'arrête jamais ou qu'elle laisse sur son ruban quelquechose qui n'a pas la structure 0^m, dans de tels cas $M(\vec{n}) \uparrow$.

Définition 3.7 *Une fonction partielle* $f : N^k \to N$ *est* Turing-semi-calculable *s'il existe une machine* M *telle que pour tous les* $\vec{n}$ *où* $f(\vec{n})$ *est définie* $M(\vec{n}) = f(\vec{n})$. *Une fonction totale* $f : N^k \to N$ *est* Turing-calculable *s'il existe une machine* M *telle que pour tout* $\vec{n}$ *on a que* $M(\vec{n}) = f(\vec{n})$.

La notion de fonction Turing-calculable (équivalente à celle de fonction récursive) formalise, à nouveau, la notion de fonction calculable, c'est dire qu'elle constitue un autre modèle mathématique de la notion intuitive de fonction pour laquelle existe un algoritme de calcul.

Exemple 3.2 *Comme premier exemple prenons une machine* M *qui calcule* $n + m$ *lorsqu'on lui donne* 0^n10^m *sur son ruban:*

$$M = (\{q_0, q_1\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B, \emptyset)$$

où δ *est donnée sous forme de table à la figure 4.*

$Q \times \Gamma$	0	1	B
q_0	$(q_0, 0, \rhd)$	$(q_0, 0, \rhd)$	$(q_1, B, \lhd)$
q_1	$(q_1, B, \bigtriangledown)$	-	-

Figure 4: la fonction δ

Exemple 3.3 *Prenons comme autre exemple une machine* M *qui calcule* $n \dot{-} m$ *lorsqu'on lui donne* 0^n10^m *sur son ruban:*

$$M = (\{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B, \emptyset)$$

où δ *est donnée sous forme de table à la figure 5. Voici des exemples du*

	$Q \times \Gamma$	0	1	B
initialisation	q_0	$(q_1, B, \rhd)$	$(q_5, B, \rhd)$	-
déplacement à droite	q_1	$(q_1, 0, \rhd)$	$(q_1, 1, \rhd)$	$(B, q_2, \lhd)$
$m := m-1$	q_2	$(q_3, B, \lhd)$	$(q_6, 0, \rhd)$	
déplacement à gauche	q_3	$(q_3, 0, \lhd)$	$(q_3, 1, \lhd)$	$(q_4, B, \rhd)$
$n := n-1$	q_4	$(q_1, B, \rhd)$	$(q_5, B, \rhd)$	
$m \geq n$: mise à 0	q_5	$(q_5, B, \rhd)$	$(q_5, B, \rhd)$	$(q_6, B, \rhd)$
état final	q_6	-	-	-

Figure 5: la fonction δ

calcul de M sur donnée $\omega = 0010$ (fig. 6) et $\omega = 0100$ (fig. 7):

$q_0 0010$	$\vdash Bq_1 010$	$\vdash B0q_1 10$	$\vdash B01q_1 0$	
	$\vdash B010q_1 B$	$\vdash B01q_2 0B$	$\vdash B0q_3 1B$	
	$\vdash Bq_3 01$	$\vdash q_3 B01$	$\vdash Bq_4 01$	
	$\vdash BBq_1 1$	$\vdash BB1q_1 B$	$\vdash BBq_2 1$	$\vdash BB0q_6$

Figure 6: le calcul de $M(0010)$

$q_0 0100$	$\vdash Bq_1 100$	$\vdash B1q_1 00$	$\vdash B10q_1 0$
	$\vdash B100q_1 B$	$\vdash B10q_2 0$	$\vdash B1q_3 0$
	$\vdash Bq_3 10$	$\vdash q_3 B10$	$\vdash Bq_4 10$
	$\vdash BBq_5 0$	$\vdash BBBq_5 B$	$\vdash BBBBq_6$

Figure 7: le calcul de $M(0100)$

3.2 Constructions de machines de Turing

Afin de pouvoir construire de façon plus efficace des machines de Turing nous allons étudier un certain nombre de techniques nous permettant de simplifier nos constructions.

mémoire locale finie

Une machine de Turing peut enregistrer localement un nombre fini de symboles de Γ en augmentant son nombre d'états. Soit Q un ensemble d'états possibles d'une machine M. On crée l'ensemble Q' des super-états de M en posant $Q' = Q \times \underbrace{\Gamma \times \Gamma \times \ldots \times \Gamma}_{k}$. Chaque super-état de Q' permet d'une part de mémoriser un état $q \in Q$ mais d'autre part de mémoriser jusqu'à k éléments de Γ.

Exemple 3.4 *Considérons le langage*

$$L = \{\alpha\beta \mid \alpha \in \Sigma, \beta \in \Sigma^* \text{et } \alpha \text{ apparaît dans } \beta\}.$$

On construit une machine acceptant L en mémorisant le premier caractère lu sur le ruban et ensuite en le cherchant dans le reste du mot donné.

$$M = (\{q_0, q_1\} \times \Gamma, \Gamma - \{B\}, \Gamma, \delta, (q_0, B), B, \{q_0\} \times \Sigma)$$

où $\delta((q_0, B), \sigma) = ((q_1, \sigma), \sigma, \rhd)$ *pour* $\sigma \in \Sigma$ *et*

$$\delta((q_1, \sigma), \sigma') = \begin{cases} ((q_0, \sigma), \sigma', \rhd) & si\ \sigma' = \sigma \\ ((q_1, \sigma), \sigma', \rhd) & si\ \sigma' \neq \sigma \end{cases}$$

Cette mémoire permet, entre autres, à nos machines de faire des décalages à droite ou à gauche (quand cela est possible) du mot en entrée.

ruban multi-pistes

Une machine de Turing peut utiliser un ruban contenant plusieurs pistes afin de pouvoir travailler sur une de ces pistes sans pour autant modifier l'information emmagasinée sur les autres. Pour ce faire, on modifie Γ afin de pouvoir conserver plusieurs symboles à la fois sur le ruban. On crée l'ensemble Γ' des supersymboles de M en posant $\Gamma' = \underbrace{\Gamma \times \Gamma \times \ldots \times \Gamma}_{k}$. Chaque supersymbole permet mémoriser k éléments de Γ dans une case du ruban.

Exemple 3.5 *Considérons par exemple le langage*

$$L = \{\omega \in \{0, 1\}^* \mid \omega \text{ est premier lorsqu'interprété en base } 2\}.$$

Il est conceptuellement beaucoup plus simple de concevoir une solution en utilisant un ruban avec trois bandes. Sur la première bande on met l'entrée ω que l'on transforme d'abord en $\&\omega\$$ afin de pouvoir facilement identifier le début et la fin de l'entrée (on peut faire une telle transformation très facilement en utilisant la mémoire locale finie). L'algorithme pour décider de la primalité du nombre en entrée est d'écrire sur la seconde piste les nombres binaires $2, 3, ..., \omega-1$ *successivement et pour chacun de ces diviseurs potentiels d, effectuer sur une troisième piste la division euclidienne de ω par d, par exemple en retranchant d autant de fois que possible de ω. ω est premier si aucun des restes obtenus n'est nul. On voit un exemple de ce procédé à la*

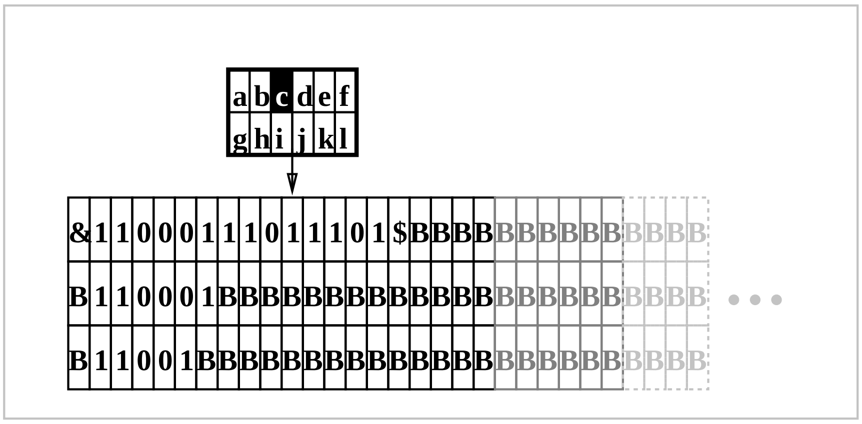

Figure 8: Machine de Turing avec ruban 3-pistes

figure ci-dessus.

marquage des cases du ruban

Une machine de Turing peut marquer les cases qu'elle a déjà lues à l'aide du symbole "$\surd$" et ainsi ne considérer que les cases non encore traités. Pour ce faire on remplace l'ensemble des symboles de travail Γ de M par $\Gamma \times \{\surd, B\}$. Ceci équivaut à ajouter une piste au ruban, laquelle contient B ou $\surd$. Cette technique est très utile pour construire des machines acceptant des langages tels que $\{\omega\omega \mid \omega \in \Sigma^*\}$.

$(q_0,B)0110$	$\vdash 0_{\surd}(q_1,0)110$	$\vdash 0_{\surd}1(q_1,0)10$	$\vdash 0_{\surd}11(q_1,0)0$
	$\vdash 0_{\surd}110(q_1,0)$	$\vdash 0_{\surd}11(q_2,0)0$	$\vdash 0_{\surd}1(q_2,B)10_{\surd}$
	$\vdash 0_{\surd}(q_2,B)110_{\surd}$	$\vdash (q_2,B)0_{\surd}110_{\surd}$	$\vdash 0_{\surd}(q_0,B)110_{\surd}$
	$\vdash 0_{\surd}1_{\surd}(q_1,1)10_{\surd}$	$\vdash 0_{\surd}1_{\surd}1(q_1,1)0_{\surd}$	$\vdash 0_{\surd}1_{\surd}(q_2,1)10_{\surd}$
	$\vdash 0_{\surd}(q_2,B)1_{\surd}1_{\surd}0_{\surd}$	$\vdash 0_{\surd}1_{\surd}(q_0,B)1_{\surd}0_{\surd}$	$\vdash 0_{\surd}1_{\surd}(q_3,B)1_{\surd}0_{\surd}$

Figure 9: le calcul de $M(0110)$

Exemple 3.6 *Prenons par exemple une machine capable d'accepter les mots de la forme $\omega\bar{\omega} \mid \omega \in \Sigma^*$ ($\bar{\omega}$ est l'image miroir de ω).*

$$M = (\{q_0,q_1,q_2,q_3\} \times \Gamma, \Gamma - \{B\} \times \{B\}, \Gamma \times \{B,\surd\}, \delta, (q_0,B), B, (q_3,B))$$

Dans l'état (q_0,B), M mémorise et marque par $\surd$ le premier symbole non-marqué.

$$\delta((q_0,B),(\sigma,B)) = ((q_1,\sigma),(\sigma,\surd),\ \rhd)$$

pour $\sigma \in \Sigma$. Ensuite dans l'état q_1 elle cherche le premier symbole à droite qui soit B ou qui soit marqué.

$$\delta((q_1,\sigma),(\gamma,m)) = \begin{cases} ((q_1,\sigma),(\gamma,B),\ \rhd) & si\ \gamma \neq B\ \ et\ m = B \\ ((q_2,\sigma),(\gamma,\surd),\lhd) & si\ \gamma = B\ \ ou\ m = \surd \end{cases}$$

Si le symbole immédiatement à sa gauche est le même que celui mémorisé, elle le marque et répète, sinon elle s'arrête en n'acceptant pas le mot.

$$\delta((q_2,\sigma),(\sigma,B)) = ((q_2,B),(\sigma,\surd),\lhd)$$

$$\delta((q_2,B),(\sigma,B)) = ((q_2,B),(\sigma,B),\lhd)$$

$$\delta((q_2,B),(\sigma,\surd)) = ((q_0,B),(\sigma,\surd),\ \rhd)$$

Finalement, M accèpte si dans l'état (q_0,B) elle rencontre un symbole marqué:

$$\delta((q_0,B),(\sigma,\surd)) = ((q_3,B),(\sigma,\surd),\bigtriangledown)$$

Ci-dessus, un exemple d'exécution avec donnée 0110; on identifie $(0,B)$ et $(1,B)$ à 0 et 1, et $(0,\surd)$ et $(1,\surd)$ à $0_{\surd}$ et $1_{\surd}$.

sous-routines

Si l'on dispose d'une machine M calculant une fonction f on peut l'utiliser dans la construction d'une autre machine. Supposant que M est décrite à l'aide d'un ruban à k bandes, pour utiliser M dans une autre machine M' il suffit de réserver k bandes de M', d'y inscrire la donnée à traiter par M puis d'écrire une copie de M qui ne considère que ces bandes. Pour faire l'appel à M il suffit de passer dans son l'état initial. Pour le retour de sous-routine, il suffit d'ecrire M de façon à ce qu'il n'y ait qu'un seul état final et que cet état rende le controle à M'.

3.3 variations sur la machine de Turing

Afin de simplifier encore plus la construction de machines, nous voyons maintenant un certain nombre de machines semblables au modèle standard mais différentes de façon non-triviale. On montre que chacun de ces modèles est équivalent au modèle standard.

ruban infini bidirectionel

Au lieu de supposer que le ruban possède une case de début comme dans le modèle standard, on suppose ici que le ruban de la machine s'étend à l'infini dans les deux directions.

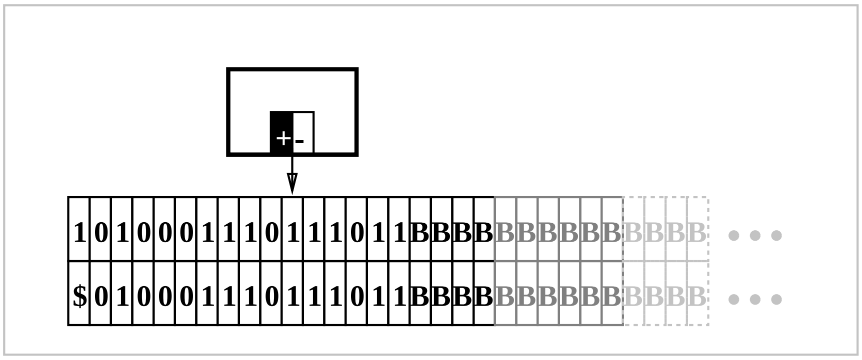

Figure 10: simulation d'un ruban infini bidirectionel

D'une part il est évident que tout calcul se faisant dans le modèle standard se fait aussi dans ce modèle, puisqu'il suffit de définir une case "départ" et de ne jamais la dépasser à sa gauche. Par contre, pour simuler le modèle bidirectionel avec le modèle standard on utilise un ruban à deux pistes. On appèle la case où se trouve la machine bidirectionelle "0", et on numérote les cases de droite par $1, 2, 3...$, et celles de gauche $-1, -2, -3, ...$. On simule le ruban bidirectionel en mettant sur la première piste les symboles en position positive et sur la seconde piste les symboles en position négative. La machine se rappelle du côté considéré au moyen de deux états "+" et "-".

Plusieurs têtes et rubans

De même que pour les machines multi-pistes, on peut définir un modèle avec plusieurs rubans disjoints et plusieurs têtes de lectures indépendantes sur chaque ruban. On désigne un de ces rubans comme celui où le mot en donnée doit être écrit au début du calcul. Pour ce type de machine, la fonction de transition δ a pour domaine $Q \times \Gamma^k$ (s'il y a k rubans) et donne comme résultat un élément de $Q \times \Gamma^k \times \{ \triangleright, \triangleleft, \bigtriangledown\}^k$. Les mouvements des têtes de lecture sont donc supposés "simultanés".

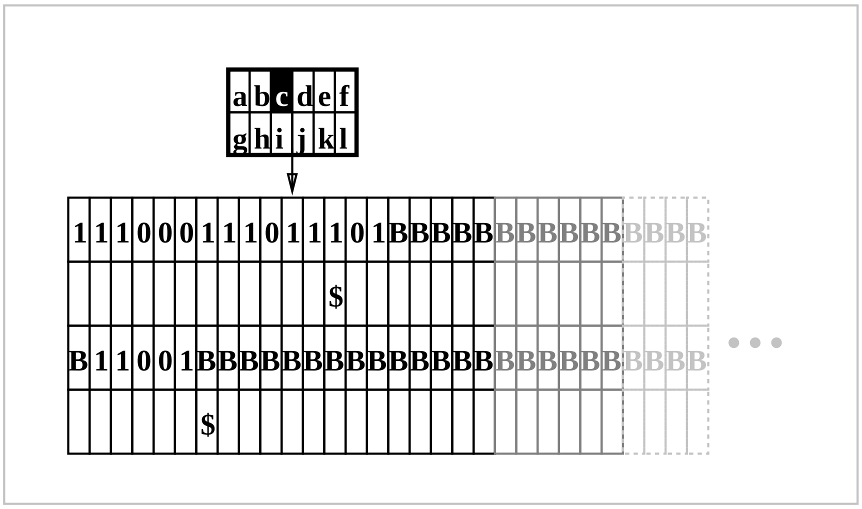

Figure 11: simulation de 2 rubans avec 2 têtes de lectures

Sans entrer trop dans les détails, l'idée de la simulation de telles machines réside simplement dans la création d'une machine multi-pistes où certaines pistes servent à simuler les rubans multiples, tandis que d'autres pistes servent à mémoriser la position des têtes multiples de lectures. Vous pouvez imaginer le reste...

Machines non-déterministes

Une machine de Turing est *non-déterministe* si la fonction de transition δ est remplacée par une relation $\delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{ \rhd, \lhd, \bigtriangledown \})$. C'est-à-dire que pour $q \in Q$ et $\gamma \in \Gamma$ il se peut que $\delta(q, \gamma)$ ait plusieurs valeurs possibles (mais toujours en nombre fini!). Pour cette raison, il se peut que pour une configuration donnée $\alpha_1 q \alpha_2$ il existe plusieurs $\alpha_3 q' \alpha_4$ tels que $\alpha_1 q \alpha_2 \vdash_M \alpha_3 q' \alpha_4$.

Néanmoins, on définit encore le langage accepté par une machine non-déterministe M par:

$$L(M) = \{\omega \mid \omega \in \Sigma^* \text{ et } q_0\omega \vdash_M^* \alpha_1 q \alpha_2 \text{ pour un } q \in F \text{ et des } \alpha_1, \alpha_2 \in \Gamma^*\}$$

mais en comprenant bien que maintenant l'acceptation de w révient à l'*existence d'au moins un calcul* qui mène de $q_0\omega$ à une configuration $\alpha_1 q \alpha_2$ où $q \in F$.

Théorème 3.8 *S'il existe une machine non-déterministe M acceptant un langage L alors il existe aussi une machine déterministe M' acceptant le même langage.*

Ébauche de preuve
L'idée intuitive est que l'ensemble des configurations de M pour une donnée w peut être structuré comme un arbre dont la racine est la configuration initiale et les fils d'un noeud correspondant à une configuration d sont toutes les configurations qui découlent de d; chaque branche correspond alors à un calcul possible de M pour la donnée w et chaque branche se terminant avec une configuration acceptante représente une suite de choix qui mène M à accepter w. On peut alors construire une machine déterministe M' qui explore cet arbre en largeur en cherchant une branche acceptante.

Un peu plus précisément, on utilise une machine à 3 bandes: sur l'une d'elles on reproduit la table de transition de la machine non déterministe (M). Sur une deuxième bande on trouvera la séquence (finie) des configurations de M accessibles en n mouvements. Initialement cette bande contient la configuration initiale de M. La troisième bande sert à simuler les transitions possibles de M à partir d'une des configurations décrites sur la deuxième bande. Une fois cette simulation terminée, on recopie sur la deuxième bande le contenu de la troisième qui contient les configurations de M accessibles en $n + 1$ mouvements. On itère cette opération jusqu'à ce qu'on obtienne une configuration acceptante de M parmi celles qui sont calculées.

■

Nombre limité de symboles ou d'états

Théorème 3.9 (3 états)
Pour toute machine de Turing $M = (Q, \Sigma, \Gamma, q_0, F, B, \delta)$ *il existe une machine* $M' = (\{q_0, q_1, q_2\}, \Sigma, \Gamma', q_0, F', B, \delta')$ *telle que* $L(M) = L(M')$.

Théorème 3.10 (3 symboles)
Pour toute machine de Turing $M = (Q, \{0, 1\}, \Gamma, q_0, F, B, \delta)$ *il existe une machine* $M' = (Q', \{0, 1\}, \{0, 1, B\}, q_0, F', B, \delta')$ *telle que* $L(M) = L(M')$.

3.4 Exercices

3.4.1

Donner une M.T. M, telle que
$L(M) = \{1^a 0 1^b 0 1^c\}$ avec $a + b = c$.

3.4.2

Donner une M.T. M, telle que
$L(M) = \{ww \mid w \in \{0, 1\}^*\}$

3.4.3

Donner une M.T. M qui calcule la fonction f qui à un mot $w \in \Sigma^*$ associe le mot ww.

3.4.4

1. Construire une machine de Turing déterministe M qui s'arrête toujours et telle que

$$L(M) = \{w\#w' \mid w \in \{a, b\}^*, w' \in \{a, b, c\}^* \text{et } w \text{ est identique à } w' \text{ si on oublie les } c \text{ de } w'\}$$

Exemples : $aba\#accbac \in L(M)$ et $ab\#aacb \notin L(M)$.

2. Construire une machine de Turing non déterministe M' qui s'arrête toujours et telle que

$$L(M') = \{w\#w' \mid w, w' \in \{a,b\}^* \text{et } w \text{ divise } w' \}$$

où, par définition, un mot w *divise* un mot w' si on peut obtenir w à partir de w' en effaçant un nombre quelconque de symboles de w'.
Exemples : aaa divise $babbaab$ et ne divise pas $abbbab$.

3.4.5

Donner une M.T. M, telle que
L(M) = { ($\#1^{a_1}0\ldots01^{a_k}01^c$: $k \geq 0$ et $\exists I \sum_{i\in I} a_i = c$ }

3.4.6

Soit Σ un alphabet fini. Montrer qu'il existe une machine de Turing M qui énumère tous le mots de Σ^*, en commencant par les mots les plus courts.

3.4.7

Montrer que n'importe quelle machine de Turing dont l'alphabet d'entrée est $\{1\}$ peut être simulée par une machine de Turing dont l'alphabet d'entrée est $\{1\}$ et l'alphabet de travail est $\{1, B\}$.

3.4.8

On appelle Score la fonction des entiers naturels dans eux-memes qui à un entier n associe le "score" maximal d'une machine de Turing sur l'alphabet $\{1\}$, à ruban bi-infini initialement vide et comportant au plus n états de travail + l'état final. Le "score" d'une telle machine est le nombre de 1 inscrits sur le ruban quand la machine s'arrête. Si la machine ne s'arrête pas, son score est 0 par convention.

1. Montrer que Score est totale et n'est pas Turing-calculable

2. Calculer Score(2).

3. Montrer que $\mathsf{Score}(3) \geq 6$. En fait, on sait que $\mathsf{Score}(3) = 6$, $\mathsf{Score}(4)$ $=13$ (mais il n'est pas demandé de le prouver). La valeur exacte de $\mathsf{Score}(5)$ est actuellement inconnue.

3.4.9

Quelles restrictions faut-il ajouter à la définition des machines de Turing pour retrouver les automates (de mot) d'états finis ?

3.4.10

Une Pile est un ruban semi-infini muni d'une tête de lecture, avec la propriété particulière que, lorsque la tête se déplace vers l'origine (à gauche), le contenu du ruban à droite de la tête est effacé. De plus, on suppose qu'il est possible de tester si la pile est vide (par exemple, en utilisant un symbole spécial de fond de pile).

1. Comment simuler le fonctionnement des automates à pile (de mots) à l'aide d'une machine de Turing ?

2. Montrer qu'on peut simuler n'importe quelle machine de Turing à l'aide d'une machine à deux piles. (i.e. une Machine de Turing à exactement deux rubans qui sont des piles).

3.4.11

Une **machine de Minski** peut être vue comme un automate à n piles sur un alphabet à une lettre (plus le fond de pile plus le blanc) . On peut alors voir les piles comme des entiers naturels (en base 1). Les machines de Minski sont donc constituées de *registres* $r_1, \ldots, r_n$ contenant des entiers naturels arbitraires.

Plus formellement, les machines de Minski sont des quadruplets (q_0, Q, δ, Q_f) où Q est un ensemble d'états, $Q_f \subseteq Q$ est l'ensemble des états finaux, $q_0 \in Q$ est l'état initial et δ est une fonction (de transition) de Q dans $(Q \cup Q \times \{1, \ldots, n\}) \cup (Q \times Q \times \{1, \ldots, n\})$

Une configuration de la machine est la donnée d'un état et de n entiers naturels $r_1, \ldots, r_n$. Par convention, initialement, r_1 contient la donnée de la fonction et r_1 contient le résultat obtenu quand la machine s'arrête. Les transition s'effectuent comme suit: si la machine est dans l'état q alors

- ou bien $\delta(q) = q'$ et la machine passe dans l'état q' sans changer le contenu des registres,
- ou bien $\delta(q) = (q', i)$ et la machine passe dans l'état q' sans changer $r_1, \ldots, r_n$, excepté r_i qui est remplacé par $1 + r_i$
- ou bien $\delta(q) = (q_1, q_2, i)$ et
  - ou bien $r_i = 0$ et la machine passe dans l'état q_1, sans changer le contenu des registres
  - ou bien $r_i > 0$ et la machine passe dans l'état q_2 sans changer le contenu des registres, excepté r_i qui est remplacé par $r_i - 1$.

La machine *calcule* $f(n)$ si, à partir d'un entier n quelconque, la machine s'arrête dans un état final avec $f(n)$ dans le registre r_1.

1. Donner une machine de Minski M_k qui calcule le quotient et le reste de la division par k (non nul) d'un entier n. (Ici, comme le résultat est une paire d'entiers, on supposera le quotient dans r_1 et le reste dans r_2.)
2. Donner pour tous entiers $b, c \geq 2$ une machine de Minski $M_{b,c}$ qui, à l'entier $N = \sum_{i=0}^{n} a_i b^i$ associe $M = \sum_{i=0}^{n} a_i c^i$, $a_1, \ldots, a_n$ étant des entiers Strictement inférieurs à b et à c.
3. Montrer que tout automate à (une) pile peut être simulé par une machine de Minski. On en déduit que les machines de Turing peuvent être simulées par une machine de Minski. (En fait, 2 registres suffisent, mais la preuve n'est pas demandée).

3.4.12 *

Montrer qu'avec une machine déterministe à 3 états on peut simuler n'importe quelle MT. (On pourra tout d'abord montrer ce résultat avec k états, k fixé au lieu de 3. Les intermédiaires $k = 4, 5$ peuvent être utiles.)

4 Le lambda-calcul pur

Le λ-calcul a été inventé en 1930 par A. Church, autour d'une notation spécifique permettant de bien exprimer des fonctions mathématiques. Considérons par exemple l'expression mathématique $x + y$; elle peut être interprétée de plusieurs façons :

- comme le nombre $x + y$
- comme une fonction f de l'argument x, c'est à dire $f : x \mapsto x + y$
- comme une fonction g de l'argument y, c'est à dire $g : y \mapsto x + y$
- comme une fonction h de 2 arguments, x et y.

La λ-notation permet d'exprimer syntaxiquement la différence entre ces quatre lectures differentes : [1]

- le nombre $x + y$ s'écrit toujours $x + y$
- la fonction f s'écrit $\lambda x.x + y$
- la fonction g s'écrit $\lambda y.x + y$
- la fonction h s'écrit $\lambda x.\lambda y.x + y$

Le λ-calcul étant un outil qui permet d'exprimer des fonctions mathématiques, les objets sur lesquels s'appliquent ces fonctions sont aussi des fonctions.

Elles peuvent être soit des fonctions "quelconques" (on parlera alors de λ-calcul non-typé) ou bien des fonctions de type donné (on parlera alors de λ-calcul typé). Le λ-calcul a de nombreuses applications au niveau informatique :

- le langage de programmation LISP est fondé sur le λ-calcul non-typé
- le langage de programmation ML est fondé sur le λ-calcul typé
- le λ-calcul est utilisé, entre autres, pour exprimer la sémantique opérationnelle de langages de programmation et pour la conception de preuves de correction de programmes, etc.

---

[1] En realité, on n'a même pas besoin du symbole + dans la syntaxe; comme on va le voir, les symboles λ, ., (,) suffisent à exprimer les fonctions calculables des entiers non-négatifs.

4.1 Substitution et Conversion

Les expressions du λ-calcul sont des suites finies formées des symboles

$$(\ ) \ \lambda \ .$$

et de variables $a, b, c, ..., a', b', c', .., a'', ...$ On compose une expression du λ-calcul (on dit une λ-expression) en utilisant un certain nombre de fois les règles suivantes:

1. une variable x est une λ-expression.
2. si E_1 et E_2 sont des λ-expressions alors $(E_1)E_2$ l'est aussi. (*Application*).
3. si x est une variable et E une λ-expression alors $\lambda x.E$ est une λ-expression. (*Abstraction*).

La λ-expression $(E_1)E_2$ exprime l'application de la fonction E_1 à E_2. La λ-expression $\lambda x.E$ exprime l'opération de construction d'une nouvelle fonction à partire de la fonction representée par E en "faisant abstraction" de x, qui devient un argument de la nouvelle fonction; si on considére à nouveau notre exemple de départ, la fonction h peut être vue comme obtenue par abstraction à partir de la fonction g.

Définition 4.1 *On définit les* **occurrences libres** *d'une variable x dans une λ-expression E par:*

- *si E est la variable x, alors l'occurrence de x dans E est libre.*
- *si $E = (E_1)E_2$, les occurrences libres de x dans E sont les occurrences libres de x dans E_1 et E_2.*
- *si $E = \lambda y.E'$, les occurrences libres de x dans E sont celles de x dans E' si $x \neq y$, sinon x n'a pas d'occurrence libre dans E.*

Une **variable libre** de E est une variable ayant au moins une occurrence libre dans E. Une λ-expression sans variable libre est dite **close**. Une **variable liée** de E est une variable apparai ssant immédiatement à droite du symbole "λ" dans E. Les λ-expressions closes n'ont donc que des variables liées. Intuitivement, on peut identifier une λ-expression E à un programme; les variables libres de E correspondent alors aux variables globales et les variables liées aux variables locales.

Exemple 4.1 *Dans les λ-expression suivantes, x est libre:*

1. $\lambda f.(f)x$

2. $(x)\lambda x.(f)x$

3. $(\lambda x.x)\lambda f.(f)x$

D'autre part, x est liée dans les expressions 2 et 3, tandis qu'elle ne l'est pas dans 1.

On voit donc dans cet exemple qu'une variable peut être en même temps libre et liée dans la même expression.

Substitution

Soit x une variable et E, X des λ-expressions. L'idée intuitive de la substitution est de former E' en remplaçant chaque occurrence libre de x dans E par la λ-expression X. On définit $E' = [X/x]E$, la substitution de x par X dans E par:

1. si E est une variable alors $E' = \begin{cases} X & \text{si } E = x \\ E & \text{si } E \neq x \end{cases}$.

2. si E est de la forme $(E_1)E_2$ alors $E' = ([X/x]E_1)[X/x]E_2$.

3. si E est de la forme $\lambda y.Y$ alors

$$E' = \begin{cases} E & \text{si } x \text{ n'est pas libre dans } E \\ \lambda y.[X/x]Y & \text{si } x \text{ est libre dans } E \text{ mais } y \text{ pas libre dans } X \\ \lambda z.[X/x][z/y]Y & \text{si } x \text{ est libre dans } E \text{ et } y \text{ libre dans } X \end{cases}$$

   où z est une variable n'ayant pas d'occurrence dans $(X)Y$.

La troisième clause de cette définition nécessite quelque commentaire; plus précisement, on va expliquer pourquoi dans le cas où x est libre dans E et y libre dans X, il faut renommer y en z. Considérons par exemple les deux λ-expressions $\lambda y.x$ et $\lambda w.x$; intuitivement, elles expriment la même fonction (et, en effet, elles sont formellement "identifiées" en λ-calcul : voir après le paragraphe sur la α-équivalence). La fonction qui est répresentée, dans les deux cas, est la fonction constante qui associe à n'importe quel argument une même valeur x. Donc, normalement, si on remplace dans les deux expressions la variable libre x par une même expression X, on doit obtenir deux nouvelles expressions qui, à nouveau, ont la même signification. Toutefois, soit $X = y$; *si on n'avait pas le renommage* correspondant au dernier cas de la définition de substitution, on aurait :

- $[y/x]\lambda y.x = \lambda y.y$, la fonction d'identité qui associe à n'importe quel argument soi même
- $[y/x]\lambda w.x = \lambda w.y$, la fonction constante qui associe à n'importe quel argument une même valeur y.

Par contre, si on fait la substitution comme indiqué par la définition on obtient :

- $[y/x]\lambda y.x = \lambda z.[y/x][z/y]x = \lambda z.[y/x]x = \lambda z.y$
- $[y/x]\lambda w.x = \lambda w.[y/x]x = \lambda w.y$

avec $\lambda z.y$ et $\lambda w.y$ qui répresentent la même fonction.

Voyons d'autre exemples de substitution :

Exemple 4.2 *que vaut* $[\lambda f.\lambda x.(f)x/x]\lambda f.\lambda y.(f)(x)y$*?*

$$\begin{aligned}
[\lambda f.\lambda x.(f)x/x]\lambda f.\lambda y.(f)(x)y &= \lambda f.[\lambda f.\lambda x.(f)x/x]\lambda y.(f)(x)y \\
&= \lambda f.\lambda y.[\lambda f.\lambda x.(f)x/x](f)(x)y \\
&= \lambda f.\lambda y.([\lambda f.\lambda x.(f)x/x]f)[\lambda f.\lambda x.(f)x/x](x)y \\
&= \lambda f.\lambda y.(f)([\lambda f.\lambda x.(f)x/x]x)[\lambda f.\lambda x.(f)x/x]y \\
&= \lambda f.\lambda y.(f)(\lambda f.\lambda x.(f)x)y
\end{aligned}$$

Exemple 4.3 *que vaut* $[\lambda x.(f)x/x]\lambda f.\lambda y.(f)(x)y$*?*

$$\begin{aligned}
[\lambda x.(f)x/x]\lambda f.\lambda y.(f)(x)y &= \lambda a.[\lambda x.(f)x/x][f/a]\lambda y.(f)(x)y \\
&= \lambda a.[\lambda x.(f)x/x]\lambda y.[f/a](f)(x)y \\
&= \lambda a.[\lambda x.(f)x/x]\lambda y.([f/a]f)[a/f](x)y
\end{aligned}$$

$$\begin{aligned}
&= \lambda a.[\lambda x.(f)x/x]\lambda y.(a)([a/f]x)[a/f]y \\
&= \lambda a.[\lambda x.(f)x/x]\lambda y.(a)(x)y \\
&= \lambda a.\lambda y.[\lambda x.(f)x/x](a)(x)y \\
&= \lambda a.\lambda y.([\lambda x.(f)x/x]a)[\lambda x.(f)x/x](x)y \\
&= \lambda a.\lambda y.(a)([\lambda x.(f)x/x]x)[\lambda x.(f)x/x]y \\
&= \lambda a.\lambda y.(a)(\lambda x.(f)x)y
\end{aligned}$$

α-conversion

On a déjà observé que $\lambda y.x$ et $\lambda w.x$ dénotent la même fonction. En général, deux λ-expressions qui ne diffèrent que par les noms de leurs variables liées peuvent être identifiées, de la même façon que, dans le calcul des prédicats, on identifie (sémantiquement), par exemple, $\forall x\ p(x,y)$ et $\forall z\ p(z,y)$. Bien entendu, pour avoir une identification il faut que le renommage préserve le caractère libre/liée de chaque variable. Par exemple, les λ-expressions $\lambda x.\lambda y.(x)y$ et $\lambda a.\lambda b.(a)b$ expriment la même fonction, tandis que $\lambda x.\lambda y.(x)y$ et $\lambda a.\lambda a.(a)a$ expriment deux fonctions différentes. Ces considérations nous mènent à la définition suivante.

Définition 4.2 (α-conversion et α-congruence) *(i) Si y n'est pas libre dans $\lambda x.X$, alors on peut réécrire $\lambda x.X$ en $\lambda y.[y/x]X$, ce que l'on note par*

$$\lambda x.X \mapsto_\alpha \lambda y.[y/x]X$$

On dit alors que $\lambda y.[y/x]X$ est obtenu à partir de $\lambda x.X$ par α-conversion.

(ii) pour toute λ-expressions E et E', si E' est obtenu à partir de E par une suite (éventuellement vide) de α-conversions alors on dit que E et E' sont α-congruentes et l'on écrit :

$$E \equiv_\alpha E'.$$

L'appellation *congruence* vient du fait que $\equiv_\alpha$ constitue une relation d'équivalence sur les λ-expressions et que :

$$E \equiv_\alpha E',\ M \equiv_\alpha N \Rightarrow [M/x]E \equiv_\alpha [N/x]E'.$$

Dans la suite, on se donnera le droit de réécrire une λ-expression en une λ-expression équivalente sans même indiquer explicitement les étapes de α-conversion.

β-réduction

La relation de β-contraction, définie ci-dessous, permet d'évaluer une λ-expression de la forme $E = (\lambda x.X)Y$, c'est-à dire effectuer une étape du calcul de la fonction $E = (\lambda x.X)Y$ pour l'argument Y.

Définition 4.3 (β-contraction, β-réduction, β-équivalence)

(i) *Une* sous-expression *d'une λ-expression E est un sous-mot F de E qui est lui-même une λ-expression.*

(ii) *Un* redex *d'une λ-expression E est une sous-expression F de la forme $(\lambda x.X)Y$.*

(iii) *Le* contracté *d'un redex $(\lambda x.X)Y$ est la λ-expression $[Y/x]X$.*

(iv) *La relation de β-*contraction *$\mapsto_\beta$ associe à une λ-expression E une λ-expression E' obtenue en remplaçant un redex de E par son contracté.*

(v) *pour toutes λ-expressions E, E', si E' est obtenue à partir de E par une suite (éventuellement vide) de β-contractions et/ou α-conversions, on dit que E* se réduit *à E' et on note :*

$$E \mapsto_\beta^* E'$$

(vi) *La relation de β* équivalence *est la plus petite relation d'équivalence sur les λ-expressions qui contienne la β-contraction et la α-conversion. Elle est notée $\equiv_\beta$.*

On a déjà vu que l'on peut voir une λ-expression E comme un programme; en poursuivant cette analogie, on peut voir $(\lambda x.X)Y$ comme l'appel du programme $\lambda x.X$ sur la donnée Y. Une application de la règle β-contraction correspond alors à une étape du calcul du programme sur la donnée Y. Une réduction de E à E' correspond à plusieurs étapes de calcul.

La β-contraction permet de réduire une λ-expression E pour en obtenir une forme "plus simple". Le but du jeu est d'essayer de réduire E jusqu'à obtenir la forme la plus simple possible de E, ou, comme l'on dit la *forme normale* de E. Ceci est precisé par la définition suivante.

Définition 4.4 (forme normale)
Une λ-expression ne contenant aucun redex est dite en forme normale. *On dit que E a une forme normale ou bien que E est normalisable s'il existe une λ-expression E' en forme normale telle que $E \mapsto_\beta^* E'$.*

Remarque Faire attention à la différence entre *être* en forme normale et *avoir* une forme normale..

Exemple 4.4 *Réduisons $((\lambda x.\lambda y.(x)y)b)c$:*

$$\begin{array}{rcl} ((\lambda x.\lambda y.(x)y)b)c & \mapsto_\beta & ([b/x]\lambda y.(x)y)c \\ & = & (\lambda y.(b)y)c \\ & \mapsto_\beta & [y/c](b)y \\ & = & (b)c \end{array}$$

laquelle est en forme normale.

Exemple 4.5 *Réduisons $((\lambda x.(a)\lambda y.(x)y)b)c$:*

$$\begin{array}{rcl} ((\lambda x.(a)\lambda y.(x)y)b)c & \mapsto_\beta & ([b/x](a)\lambda y.(x)y)c \\ & = & ((a)\lambda y.(b)y)c \end{array}$$

laquelle est en forme normale.

Il n'est pas vrai que toute λ-expression est normalisable... Considérez les deux exemples suivants.

Exemple 4.6 *Réduisons $(\lambda x.(x)x)\lambda x.(x)x$:*

$$(\lambda x.(x)x)\lambda x.(x)x \quad \mapsto_\beta \quad (\lambda x.(x)x)\lambda x.(x)x$$

Puisqu'il existe une seule façon d'appliquer la règle de β-contraction à cette λ-expression, on en conclut que $(\lambda x.(x)x)\lambda x.(x)x$ n'a pas de forme normale.

Exemple 4.7 *Soit $E = \lambda x.((x)x)x$, réduisons $(E)E$:*

$$\begin{array}{rcl} (E)E & \mapsto_\beta & ((E)E)E \\ & \mapsto_\beta & (((E)E)E)E \\ & \mapsto_\beta & \ldots \end{array}$$

$(E)E$ n'a pas de forme normale pour la même raison que dans l'exemple précédent.

Unicité de la forme normale

On voudrait maintenant établir que la forme normale d'une λ-expression E est unique.

Théorème 4.5 (Church-Rosser) *Pour toutes λ-expressions E, E_1, E_2, si $E \mapsto_\beta^* E_1$ et $E \mapsto_\beta^* E_2$, alors il existe une λ-expression E_3 telle que $E_1 \mapsto_\beta^* E_3$ et $E_2 \mapsto_\beta^* E_3$.*

On peut en déduire tout d'abord une version plus constructive de la β-équivalence (la preuve sera vue en TD):

Corollaire 4.6 *$E \equiv_\beta E'$ si et seulement si il existe E_1, E_1' tels que $E \mapsto_\beta^* E_1$, $E' \mapsto_\beta^* E_1'$ et $E_1 \equiv_\alpha E_1'$.*

On peut aussi déduire du théorème de Church-Rosser l'important corollaire suivant:

Corollaire 4.7 (Unicité de la forme normale) *Soit E une λ-expression et soit N une λ-expression normale telle que $E \mapsto_\beta^* N$. Pour toute N' normale telle que $E \mapsto_\beta^* N'$ on a que $N' \equiv_\alpha N$.*

Preuve du corollaire
Soit $E \mapsto_\beta^* N$ et $E \mapsto_\beta^* N'$. Par le théorème de Church-Rosser on a :

$$\exists M (N \mapsto_\beta^* M \wedge N' \mapsto_\beta^* M).$$

Or, si N et N' sont normales, ceci veut dire qu'ils sont α-congruents à M. C'est-à-dire que N, la forme normale de E, est unique (modulo renommages de variables liées). ■

On va pousser l'analogie λ-expressions/programmes encore plus loin :

- λ-expression E = programme E
- une réduction de $(\lambda x.X)Y$ à E' = un morceau du calcul du programme $\lambda x.X$ sur la donnée Y qui méne au résultat (intermediaire) E'
- une réduction de $(\lambda x.X)Y$ à E' où E' est normale = un calcul du programme $\lambda x.X$ sur la donnée Y ayant comme résultat final E'
- une réduction de $(\lambda x.X)Y$ qui se poursuit indéfiniment (voir les deux derniers exemples) = un calcul du programme $\lambda x.X$ sur la donnée Y qui ne se termine pas
- une λ-expression $(\lambda x.X)Y$ qui n'est pas normalisable = un programme $\lambda x.X$ dont aucun calcul pour la donnée Y ne termine.

Stratégies de réduction

Il faut faire attention : le théorème de Church-Rosser avec son corollaire **ne dit pas que si E a une forme normale N alors toute réduction de E va finir par N**, car il est possible que certaines suites de β-contractions ne mènent jamais à une λ-expression normale.

Exemple 4.8 *Reprenons l'expression E de l'exemple 4.7. On sait que $(E)E$ n'a pas de forme normale. Alors considérons la λ-expression*

$$(\lambda x.\lambda y.y)(E)E$$

laquelle possède une forme normale car $(\lambda x.\lambda y.y)(E)E \mapsto_\beta^* \lambda y.y$. Néanmoins

$$\begin{array}{rll} (\lambda x.\lambda y.y)(E)E & \mapsto_\beta & (\lambda x.\lambda y.y)((E)E)E \\ & \mapsto_\beta & (\lambda x.\lambda y.y)(((E)E)E)E \\ & \mapsto_\beta & \ldots \end{array}$$

est une suite de β-contractions ne menant jamais à une forme normale.

Le théorème affirme seulement que toute suite de contractions qui se termine mènera à N (à une α-conversion près). Par bonheur, nous pouvons spécifier une suite d'applications de $\mapsto_\beta$ qui terminera dès qu'une forme normale existe. On définit la **réduction normale** d'une λ-expression comme celle qui applique toujours la β-contraction au rédex le plus à gauche.

Théorème 4.8 (Curry) *Soit E une λ-expression. S'il existe N en forme normale telle que $E \mapsto_\beta^* N$ alors la réduction normale de E mène aussi à N.*

Intuitivement, ceci veut dire que la forme normale de E existe (E est normalisable) si et seulement si la réduction normale de E se termine.

4.2 Représentation des fonctions entières

Nous allons maintenant nous consacrer à la description des fonctions entières en termes de λ-calcul. Tout d'abord, rappelons nous que le λ-calcul ne traîte que des fonctions. Il nous faut donc définir les entiers dans le λ-calcul sous forme de fonctions.

Entiers et fonctions entières

Définition 4.9 *On définit les entiers non-négatifs comme suit:*

$$
\begin{aligned}
&\overline{0} = \lambda x.\lambda y.y \\
&\overline{1} = \lambda x.\lambda y.(x)y \\
&\overline{2} = \lambda x.\lambda y.(x)(x)y \\
&\overline{3} = \lambda x.\lambda y.(x)(x)(x)y \\
&\ldots \\
&\overline{n} = \lambda x.\lambda y.\underbrace{(x)(x)(x)\ldots(x)(x)(x)}_{n}y \\
&\ldots
\end{aligned}
$$

On appelle ces fonctions les **entiers de Church**. Intuitivement, l'entier n est répresenté comme un "iterateur", plus exactement comme une fonction à deux arguments x et y qui applique la fonction x à la donnée y, puis applique x une deuxième fois au résultat obtenu,...puis applique x une n-ème fois au dernier résultat obtenu.

On peut maintenant définir des fonctions sur N. L'évaluation de ces fonctions va se faire au moyen de la β-contraction. Remarquez que tous les entiers de Church sont des λ-expressions en forme normales. Donc pour évaluer une fonction F sur la donnée $\overline{n}$, on va procéder à la réduction normale de l'expression $(F)\overline{n}$.

Exemple 4.9 *On définit la λ-expression $SUC = \lambda x.\lambda y.\lambda z.(y)((x)y)z$ qui a la propriété de transformer l'entier de Church $\overline{n}$ en $\overline{n+1}$. Voici comment:*

$$\begin{array}{rcl}
(SUC)\overline{n} & = & (\lambda x.\lambda y.\lambda z.(y)((x)y)z)\lambda x.\lambda y.\underbrace{(x)(x)(x)...(x)(x)(x)}_{n}y \\
& \mapsto_\beta & \lambda y.\lambda z.(y)((\lambda x.\lambda y.\underbrace{(x)(x)(x)...(x)(x)(x)}_{n}y)y)z \\
& \mapsto_\beta & \lambda y.\lambda z.(y)(\lambda a.\underbrace{(y)(y)(y)...(y)(y)(y)}_{n}a)z \\
& \mapsto_\beta & \lambda y.\lambda z.(y)\underbrace{(y)(y)(y)...(y)(y)(y)}_{n}z \\
& \mapsto_\beta & \lambda y.\lambda z.\underbrace{(y)(y)(y)...(y)(y)(y)}_{n+1}z \\
& \mapsto_\alpha & \overline{n+1}
\end{array}$$

Il est également possible de définir de nombreuse fonctions à valeurs entières de $\bigcup_{k\in N}\{f : N^k \to N\}$ à l'aide du λ-calcul. Une fonction de $N^k \to N$ sera représentée par une expression de la forme $\underbrace{\lambda a.\lambda a'....\lambda a^{(k-1)}.}_{k}E$ et évaluée sur les k arguments $\overline{n_1}, \overline{n_2}, ..., \overline{n_k}$ en faisant la réduction normale de l'expression:

$$(...(((\underbrace{\lambda a.\lambda a'....\lambda a^{(k-1)}.}_{k}E)\overline{n_1})\overline{n_2})...)\overline{n_k}$$

Exemple 4.10 *On définit la λ-expression $ADD = \lambda x.\lambda y.\lambda a.\lambda b.((x)a)((y)a)b$ qui a la propriété de transformer les entiers de Church $\overline{n}$ et $\overline{m}$ en $\overline{n+m}$. Voici comment:*

$$\begin{array}{rcl}
((ADD)\overline{n})\overline{m} & = & ((\lambda x.\lambda y.\lambda a.\lambda b.((x)a)((y)a)b)\overline{n})\overline{m} \\
& \mapsto_\beta & (\lambda y.\lambda a.\lambda b.((\overline{n})a)((y)a)b)\overline{m} \\
& \mapsto_\beta & \lambda a.\lambda b.((\overline{n})a)((\overline{m})a)b \\
& = & \lambda a.\lambda b.((\lambda x.\lambda y.\underbrace{(x)(x)(x)...(x)(x)(x)}_{n}y)a)((\overline{m})a)b \\
& \mapsto_\beta & \lambda a.\lambda b.(\lambda y.\underbrace{(a)(a)(a)...(a)(a)(a)}_{n}y)((\overline{m})a)b
\end{array}$$

$$
\begin{aligned}
\mapsto_\beta &\quad \lambda a.\lambda b.\underbrace{(a)(a)(a)...(a)(a)(a)}_{n}((\overline{m})a)b \\
= &\quad \lambda a.\lambda b.\underbrace{(a)(a)(a)...(a)(a)(a)}_{n}((\lambda x.\lambda y.\underbrace{(x)(x)(x)...(x)(x)(x)}_{m}y)a)b \\
\mapsto_\beta &\quad \lambda a.\lambda b.\underbrace{(a)(a)(a)...(a)(a)(a)}_{n}(\lambda y.\underbrace{(a)(a)(a)...(a)(a)(a)}_{m}y)b \\
\mapsto_\beta &\quad \lambda a.\lambda b.\underbrace{(a)(a)(a)...(a)(a)(a)}_{n}\underbrace{(a)(a)(a)...(a)(a)(a)}_{m}b \\
= &\quad \lambda a.\lambda b.\underbrace{(a)(a)(a)...(a)(a)(a)}_{n+m}b \\
\mapsto_\alpha &\quad \overline{n+m}
\end{aligned}
$$

Booléens et fonctions booléennes

Afin de nous faciliter la tâche de “programmer” les fonctions récursives en λ-calcul, on se définit aussi les booléens $(VRAI, FAUX)$ et des opérations booléennes en λ-calcul.

Définition 4.10 *On pose* $FAUX = \lambda x.\lambda y.y$ *et* $VRAI = \lambda x.\lambda y.x$.

On remarque que l'on a $FAUX = 0$, mais $VRAI \neq 1$. On définit très facilement les fonctions booléennes élémentaires: NON,ET et OU par

$$
\begin{aligned}
NON &= \lambda x.((x)FAUX)VRAI \\
ET &= \lambda x.\lambda y.((x)y)x \\
OU &= \lambda x.\lambda y.((x)x)y.
\end{aligned}
$$

On peut aussi définir des fonctions ayant un sens un peu plus complexe que des simples opérations booléennes. Par exemple, posons

$$IF = \lambda x.\lambda y.\lambda z.((x)y)z$$

qui est une fonction qui reçoit une premier paramètre booléen B et ensuite deux λ-expressions E_1 et E_2, telle que la β-contraction de $(((IF)B)E_1)E_2$ va mener à E_1 si $B = VRAI$ et à E_2 si $B = FAUX$. Par la suite, pour des raisons de lisibilité, nous nous permettrons d'écrire

$$
\begin{array}{lll}
\mathsf{IF}\ B & \mathsf{THEN} & E_1 \\
 & \mathsf{ELSE} & E_2
\end{array}
$$

au lieu de $(((IF)B)E_1)E_2$.

On va également pouvoir utiliser les booléens et les fonctions booléennes dans la construction de fonctions entières.

Exemple 4.11 *On définit l'expression $ZERO? = \lambda n.((n)(VRAI)FAUX)VRAI$ qui se réduit à $VRAI$ lorsqu'évalué avec $\overline{0}$ et qui se réduit à $FAUX$ lorqu'évalué avec $\overline{n}$ pour $n > 0$. On peut définir la fonction entière SIG qui vaut $\overline{0}$ en $\overline{0}$ et $\overline{1}$ en $\overline{n}$ pour $n > 0$ ainsi*

$$SIG = \lambda n.\,\mathsf{IF}\ n = 0\ \ \begin{array}{l}\mathsf{THEN}\ \overline{0}\\ \mathsf{ELSE}\ \overline{1}\end{array}$$

si l'on prend la convention d'écrire $n = 0$ à la place de $(ZERO?)n$

Récursivité en λ-calcul

Le problème de la définition des fonctions par récurrence

Supposons que soient définies les λ-expressions $PRED$, qui représente la fonction de prédécesseur d'un entier de Church et $MULT$ qui représente la fonction de multiplication de deux entiers de Church. Supposons maintenant que l'on veuille definir une λ-expression pour la factorielle $FACT$. Considérons l'équation

$$X = \lambda n.\,\mathsf{IF}\ \ n = 0\ \ \begin{array}{l}\mathsf{THEN}\ \overline{1}\\ \mathsf{ELSE}\ \ ((MULT)n)(X)(PRED)n\end{array}$$

On souhaite évidemment que $FACT$ soit une λ-expression qui satisfait cette équation, c'est-à-dire telle que

$$FACT \equiv_\beta \lambda n.\,\mathsf{IF}\ \ n = 0\ \ \begin{array}{l}\mathsf{THEN}\ \overline{1}\\ \mathsf{ELSE}\ \ ((MULT)n)(FACT)(PRED)n\end{array}$$

Pour ce faire, posons d'abord

$$H = \lambda f.\lambda n.\,\mathsf{IF}\ \ n = 0\ \ \begin{array}{l}\mathsf{THEN}\ \overline{1}\\ \mathsf{ELSE}\ \ ((MULT)n)(f)(PRED)n\end{array}$$

Intuitivement, ce que l'on voudrait obtenir est un "point fixe" de la fonction H, c'est à dire une expression $FACT$ telle que $FACT$ soit β-équivalente à $(H)FACT$.

Le point fixe

Théorème 4.11 (Théorème du pont fixe) *Il existe une λ- expression Y telle que, pour toute λ-expression T*

$$(Y)T \equiv_\beta (T)(Y)T.$$

Preuve. Il suffit de prendre (mais ce n'est pas le seul choix possible)

$$Y = \lambda h.(\lambda x.(h)(x)x)\lambda x.(h)(x)x$$

(opérateur de Curry). En fait, soit $G = (\lambda x.(T)(x)x)\lambda x.(T)(x)x$. On a :

$$\begin{array}{rcl} (Y)T & = & \Big(\lambda h. \big(\lambda x.(h)(x)x \big) \lambda x.(h)(x)x \Big) T \\ & \mapsto_\beta & \big(\lambda x.(T)(x)x \big) \lambda x.(T)(x)x \\ & \mapsto_\beta & (T) \big(\lambda x.(T)(x)x \big) \lambda x.(T)(x)x \\ & = & (T)G. \end{array}$$

D'autre part :

$$\begin{array}{rcl} (T)(Y)T & = & (T) \Big(\lambda h. \big(\lambda x.(h)(x)x \big) \lambda x.(h)(x)x \Big) T \\ & \mapsto_\beta & (T) \big(\lambda x.(T)(x)x \big(\lambda x.(T)(x)x \\ & = & (T)G. \end{array}$$

Donc :

$$(Y)T \equiv_\beta (T)G \equiv_\beta (Y)(T)(Y).$$

■

L'opérateur de point fixe sert à assurer, étant donnée n'importe quelle fonction F en λ-calcul, l'existence d'une λ-expression E dont F est un point fixe. Il nous sera très utile pour définir en λ-calcul des fonctions par récurrence. Par exemple, reprennons notre essai d'écrire la fonction FACT. On cherchait un point fixe de H. Il suffit de lui appliquer l'opérateur Y de Curry du théorème précedant pour obtenir ce point fixe; on pose donc finalement :

$$FACT = (Y)H.$$

Cet exemple nous montre une technique générale permettant de définir récursivement des fonctions en λ-calcul. En effet, le théorème de point fixe a plusieurs applications. Par exemple, supposons qu'on ait besoin d'une λ-expression "AVALER" qui se limite à "avaler" son argument sans le toucher pour ensuite renvoyer "AVALER" lui-même. Informellement, on veut donc $AVAL = \lambda x.AVAL$. On pose alors

$$H = \lambda f.\lambda x.f$$

et l'expression cherchée est $(Y)H$, oú Y est l'opèrateur de point fixe.

4.3 Exercices

4.3.1

Que vaut $[(y)z/x]\lambda y.(x))(\lambda x.x)y$?

4.3.2

Déterminer si les λ-expressions suivantes ont une forme normale:
$E_1 = (\lambda x.(\lambda y.(y)x)z)v$
$E_2 = (\lambda x.((x)x)y)\lambda x.((x)x)y$.

4.3.3

Prouver ou réfuter les enoncés suivants:
i) $\forall\lambda$-expression E et E', si $[E/x]E'$ est en forme normale, alors E' est en forme normale.
ii) $\forall\lambda$-expression E et E', si $[E/x]E'$ a une forme normale, alors E' a une forme normale.

4.3.4

Montrer que la λ-expression $MULT = \lambda x.\lambda y.\lambda z.(x)(y)z$ fait la multiplication de deux entiers de Church.

4.3.5

Soit la λ-expression

$$
\begin{array}{rrrrr}
PRED = & \lambda n.(& (& (n) & \\
 & & & \lambda p.\lambda u.(& (u) \\
 & & & & (SUC) \\
 & & & & (p) \\
 & & & & VRAI \\
 & & &) & \\
 & & & (p) & \\
 & & & VRAI & \\
 & &) & & \\
 & & \lambda u.(& (u) & \\
 & & & \overline{0} & \\
 & &) & & \\
 & & \overline{0} & & \\
 &) & & & \\
 & FAUX & & &
\end{array}
$$

Prouver que $PRED$ calcule le prédécesseur d'un entier de Church, c'est-à-dire que, pour tout n, $(PRED)(SUCC)\overline{n} \mapsto_\beta^* \overline{n}$. Combien y a-t-il d'étapes de β-réduction pour ce calcul?

4.3.6

Une λ-expression E est dite minimale par rapport à la β-réduction ssi

$$\forall M, E \mapsto_\beta^* M \quad \Rightarrow \quad M \equiv_\alpha E$$

Prouver que toute expression normale est minimale, mais que la réciproque est fausse.

4.3.7

Déduire du théorème 4.5 vu en cours (confluence de la β-réduction) que

$$E \equiv_\beta E' \quad \Longleftrightarrow \quad \exists E_1, E \mapsto_\beta^* E_1 \wedge E' \mapsto_\beta^* E_1$$

4.3.8

Trouver une λ-expression représentant la fonction exponentielle, c'est à dire une expression EXP telle que $((EXP)n)m)$ se réduit en n^m.

4.3.9

Donner une λ-expression Y_T (combinateur de point fixe de Turing) telle que

$$(Y_T)F \mapsto_\beta^* (F)(Y_T)F$$

4.3.10

Soient les λ-expressions:
$A = \lambda x.\lambda y.(y)x$
$M = \lambda x.\lambda y(((x)A)\lambda z.1)((y)A)\lambda z.0$
$D = \lambda x.\lambda y.\lambda a.(((M)x)y)(((M)y)x)a$
Trouver les fonctions sur les entiers représentés par M et D.

4.3.11

Soit $\lambda f.((\ldots((f)\overline{n_1})\ldots)\overline{n_k})VRAI$ la représentation de la liste $< n_1, \ldots, n_k >$ d'entiers.

1. Trouver une λ-expression $VRAI?$ telle que

$$\begin{array}{rcll} (VRAI?)VRAI & \mapsto_\beta^* & VRAI & \\ (VRAI?)\overline{n} & \mapsto_\beta^* & FAUX & \quad \text{pour } n \in \mathbf{N} \end{array}$$

2. Trouver une λ-expression $NIL?$ telle que

$$\begin{array}{lcll} (NIL?)L & \mapsto_\beta^* & FAUX & \quad \text{Si } L \text{ représente une liste non vide} \\ (NIL?)L & \mapsto_\beta^* & VRAI & \quad \text{Si } L \text{ représente une liste vide} \end{array}$$

3. Trouver une λ-expression SUM telle que

$$(SUM)\lambda f.((\ldots((f)\overline{n_1})\ldots)\overline{n_k})VRAI \mapsto_\beta^* \overline{n_1 + \ldots + n_k}$$

4. Trouver des λ-expressions $CONS, CAR, CDR$

5. Trouver une λ-expression CONCAT qui fait la concaténation de deux listes d'entiers.

4.3.12

Montrer que toute expression du λ-calcul en forme normale est ou bien une variable, ou bien s'écrit

$$\lambda x_1. \ldots \lambda x_n.((\ldots(x)E_1)E_2\ldots)E_n$$

où x est une variable et $E_1, \ldots, E_n$ sont en forme normale.

4.3.13

On note **I**, **K**, **S** les termes suivants:

$$\begin{array}{rcl} \mathbf{I} & = & \lambda x.x \\ \mathbf{K} & = & \lambda x.\lambda y.x \\ \mathbf{S} & = & \lambda x.\lambda y.\lambda z.((x)z)(y)z \end{array}$$

On appelle *terme* sur X toute expression du λ-calcul construite uniquement avec **S**, **K**, des variables $x \in X$ et l'application.

1. Montrer que **I** est β-équivalent à un terme
2. Montrer que si t est un terme sur X et $x \notin X$ alors $\lambda x.t$ est β-équivalent à un terme sur X
3. montrer que si t_1, t_2 sont des termes sur X avec $x \in X$, alors $\lambda x.(t_1)t_2$ est β-équivalent à un terme sur X.
4. Est-ce que chaque expression sans variable libre du λ-calcul est β-équivalente à un terme sur l'ensemble vide? Justifier sa réponse.

4.3.14

La *β-réduction faible* est définie par:

- $(\lambda x.E)F \mapsto [F/x]E$
- Si $M \mapsto M'$ alors $(M)N \mapsto (M')N$
- Si $N \mapsto N'$ alors $(M)N \mapsto (M)N'$

Donner une expression irréductible pour la β-réduction faible, mais qui contient un redex. Montrer que la β-réduction faible n'est pas confluente.

5 Théorèmes d'équivalence

Nous allons étudier l'équivalence de trois modèles de calcul. Les deux premiers ont déjà ètè vus: les fonctions partielles récursives et les fonctions Turing-semi-calculables. Le troisième modèle est issu du λ-calcul. Commençons par le préciser.

Fonctions λ-représentables

On définit maintenant les fonctions du λ-calcul qui sont **λ-représentables**. Quand le contexte est clair on écrira $F\overline{n_1}\overline{n_2}...\overline{n_k}$ ou $F(\overline{n_1},\ldots,\overline{n_k})$ au lieu de $(...(((F)\overline{n_1})\overline{n_2})...)\overline{n_k}$.

Définition 5.1 *Soit f une fonction partielle de $N^k \to N$. On dit qu'une λ-expression F λ-représente f si*

$$\forall n, n_1, ..., n_k \left[\begin{array}{ll} F\overline{n_1}\overline{n_2}...\overline{n_k} \equiv_\beta \overline{n} & si\ f(n_1, n_2, ..., n_k) = n \\ F\overline{n_1}\overline{n_2}...\overline{n_k}\ n'est\ pas\ normalisable & si\ f(n_1, n_2, ..., n_k) \uparrow \end{array} \right]$$

Nous avons à présent à notre disposition 3 modèles de calcul: les fonctions partielles récursives, Turing semi-calculables et λ-représentables. Nous allons voir en fait ici que ces trois modèles sont "équivalents", en ce sens qu'ils ont le même pouvoir d'expression, ce qui vient à l'appui de la thèse de Church déjà énoncée. Ces résultats d'équivalence ne signifie pas pourtant que les modèles sont interchangeables. On peut en effet dire aussi que la plupart des langages de programmation ont le même pouvoir d'expression, et pourtant chaque langage a ses intérêts propres: chaque langage apporte des notations qui sont adéquates pour exprimer certains problèmes et pas d'autres. Il en va de même des modèles de calcul. Il n'existe (jusqu'à présent) pas de modèle de calcul qui permette d'exprimer mieux que tous les autres tous les problèmes. De plus, comme mous le verrons dans le chapitre 7, la notion de complexité, a priori, dépend du modèle choisi.

Pour prouver l équivalence des modèles, nous allons procéder de la façon suivante :

1. D'abord on montre que toute fonction récursive partielle est λ-représentable;

2. puis on montre que toute fonction λ-représentable est Turing-semi-calculable;

3. enfin on montre que toute fonction Turing-semi-calculable est récursive partielle.

La preuve du premier de ces points est plus aisée si on utilise une caracterisation de l'ensemble des fonction récursives partielles due à Kleene. D'abord, on définit l'ensemble des fonctions *récursives primitives* comme l'ensemble de toutes le fonctions récursives (totales) qui peuvent être définies sans jamais utiliser la minimisation.

Théorème 5.2 (Kleene) *Etant donnée une fonction récursive partielle quelconque φ, il existe des fonctions récursives primitives γ et ψ telles que*

$$\varphi(n_1, \cdots, n_k) = \gamma(\min_m[\psi(n_1, \cdots, n_k, m) = 0]).$$

Lemme 5.3 *Les fonctions $Z(n)$, $S(n)$ et Pi_i^k sont λ-représentables par les λ-expressions suivantes :*

$$Z = \lambda x.\overline{0}$$

$$S = SUC$$

$$Pi_i^k = \lambda x.\lambda x' \ldots . \lambda x^{(k-1)}.x^{(i-1)}$$

Lemme 5.4 *Si ψ (totale): $N^{k+2} \to N$ et χ (totale): $N^k \to N$ sont λ-représentables par des λ-expressions Ψ et X alors la fonction φ définie par récursion primitive à partir de χ et ψ est aussi λ-représentable.*

Preuve du lemme 5.4: Cherchons une λ-expression Φ pour λ-représenter φ. Considérons

$$\Phi = \lambda m.\lambda n_1 \ldots . \lambda n_k. \ \mathsf{IF}\ m = 0 \ \ \begin{array}{l} \mathsf{THEN}\ X(n_1, \ldots, n_k) \\ \mathsf{ELSE}\ \ E_\Phi \end{array}$$

avec

$$E_\Phi = \Psi(\Phi((PRED)m, n_1, \cdots, n_k), (PREDm), n_1, \ldots, n_k)$$

On sait comment résoudre cette équation à l'aide de l'opérateur de point fixe Y; il suffit de poser

$$H = \lambda f.\lambda m.\lambda n_1 \ldots . \lambda n_k. \ \mathsf{IF}\ m = 0 \ \ \begin{array}{l} \mathsf{THEN}\ X(n_1, \ldots, n_k) \\ \mathsf{ELSE}\ \ E_f \end{array}$$

et $(Y)H = \Phi$ pour obtenir un Φ satisfaisant. On vérifie aisément que $\Phi(m, n_1, \ldots, n_k)$ a comme forme normale $\overline{n}$ où $n = \varphi(m, n_1, \ldots, n_k)$. ■

Lemme 5.5 *Si ψ_1, ψ_2,...,ψ_m dans $N^k \to N$ et χ dans $N^m \to N$ sont totales et λ-représentables par des λ-expressions Ψ_1, Ψ_2,..., Ψ_m et X alors la composition*

$$\chi(\psi_1, \psi_2, ..., \psi_m)$$

l'est aussi.

Preuve du lemme 5.5: Il suffit de prendre l'expression

$$\lambda n_1.\lambda n_2....\lambda n_k.(...(((X)\Psi_1 n_1...n_k)\Psi_2 n_1...n_k)...)\Psi_m n_1...n_k$$

pour λ-représenter

$$\chi(\psi_1, \psi_2, ..., \psi_m).$$

■

On a donc montré le résultat suivant :

Théorème 5.6 *Les fonctions récursives primitives sont λ-représentables.*

Observons que l'expression ci-dessus ne peut pas être utilisée pour représenter la composition quand l'une ou l'autre des fonctions composantes n'est pas totale. La raison justifiant ce fait est qu'il est possible que notre λ-expression se réduise à un entier de Church alors que la fonction $\chi(\psi_1, \psi_2, ..., \psi_m)$ n'est pas définie.

Exemple 5.1 *Soit ψ indéfinie partout, zero la fonction à valeur constante 0. La fonction composée zero(ψ) est indéfinie partout. Mais, si $Z = \lambda x.\overline{0}$,*

$$(\lambda n.(Z)(\Psi)n)\overline{m}$$

a a pour forme normale $\overline{0}$.

Lemme 5.7 *Soit χ une fonction totale à $k+1$ arguments λ-représentable par une λ-expression X. Soit φ définie par minimisation de χ par rapport à son dernier argument. Il existe une λ-expression Φ qui λ-représente φ.*

Preuve du lemme 5.7 : Intuitivement, il s'agit de définir en λ-calcul un programme $AUX(n_1, \dots, n_k, m)$ qui renvoie m si $\chi(n_1, \cdots, n_k, m) = 0$ et qui renvoit $AUX(n_1, \dots, n_k, m+1)$ sinon; si on exécute $AUX(n_1, \dots, n_k, 0)$, le programme produira le premier m tel que $\chi(n_1, \cdots, n_k, m) = 0$.

- Soient :

$$\begin{array}{lll} T = \lambda x.\lambda y. & \mathsf{IF}\ y = 0 & \mathsf{THEN}\ \overline{0} \\ & & \mathsf{ELSE}\ \ \lambda u.\lambda v.(((u)(x)(SUC)v)u)(SUC)v \\ M = & ((\ ((T)a)(a)b &)(T)a)b \end{array}$$

On a les deux proprietés suivantes (preuve en exercice) :

$$\begin{array}{llll} [E_1/a, E_2/b]\ \ M & \mapsto_\beta^* & E_2 & \text{si } (E_1)E_2 \equiv_\beta \overline{0} \\ [E_1/a, E_2/b]\ \ M & \mapsto_\beta^* & [E_1/a, (SUC)E_2/b]\ \ M & \text{si } (E_1)E_2 \equiv_\beta \overline{n+1} \end{array}$$

- Soit $AUX = \lambda a_{k+1}.((\lambda a.\lambda b.M)(\cdots((X)a_1)\cdots)a_k)a_{k+1}$ et

$$\Phi = \lambda a_1.\cdots\lambda a_k.(AUX)\overline{0}.$$

On a :

$$\begin{array}{lll} (\Phi)\overline{n_1}\dots\overline{n_k} & \mapsto_\beta^* & [\overline{n_1}/a_1, \cdots, \overline{n_k}/a_k]\ \ (AUX)\overline{0} \\ & \mapsto_\beta^* & ((\lambda a.\lambda b.M)(\cdots(X)\overline{n_1})\cdots\overline{n_k})\overline{0} \\ & \mapsto_\beta^* & [(\cdots(X)\overline{n_1}\cdots)\overline{n_k}/a, \overline{0}/b]\ \ M \end{array}$$

et, par récurrence sur m,

- si $\chi(n_1, \dots, n_k, m) = 0$ et m est minimal ayant cette propriété,

$$\begin{array}{lll} [(\cdots(X\overline{n_1})\cdots)\overline{n_k}/a, \overline{0}/b]M & \mapsto_\beta^* & [(\cdots((X)\overline{n_1})\cdots)\overline{n_k}/a, \overline{m}/b]M \\ & \mapsto_\beta^* & \overline{m} \end{array}$$

- et, si $\chi(n_1, \dots, n_k, n) > 0$ pour tout $n \leq m$, alors

$$\begin{array}{lll} [(\cdots((X)\overline{n_1})\cdots)\overline{n_k}/a, \overline{0}/b]M & \mapsto_\beta^* & [(\cdots((X)\overline{n_1})\cdots)\overline{n_k}/a, \overline{m}/b]M \\ & \mapsto_\beta^* & [(\cdots((X)\overline{n_1})\cdots)\overline{n_k}/a, \overline{m+1}/b]M \end{array}$$

Ceci, grâce aux propriétés vues plus haut.

Donc $\Phi\overline{n_1}\cdots\overline{n_k} \mapsto_\beta^* \overline{m}$ lorsque m est le plus petit entier tel que $\chi(n_1,\ldots,n_k,m)=0$. D'autre part, lorsqu'un tel entier n'existe pas, la réduction la plus à gauche de $\Phi\overline{n_1}\cdots\overline{n_k}$ ne termine pas, comme le montre la dernière réduction ci-dessus. ■

En utilisant le lemme ci-dessus, on peut établir le théoreme que nous interesse.

Théorème 5.8 *Les fonctions récursives partielles sont λ-représentables*

Preuve. Par le résultat de Kleene, toute fonction récursive partielle ψ peut s'écrire :

$$\psi(n_1,\cdots,n_k)=\gamma(\min_m[\chi(n_1,\cdots,n_k,m)=0]).$$

où γ et χ sont récursives primitives, donc λ-représentables par des expressions X et Γ. Par le lemme 5.7, la fonction

$$\varphi(n_1,\ldots,n_k)=\min_m[\chi(n_1,\cdots,n_k,m)=0]$$

est λ-représentable par une λ-expression Φ.

Soit $\Psi^*=\lambda a_1.\cdots\lambda a_k.(\Gamma)\Phi a_1\cdots a_k$. Il est clair que si $\psi(n_1,\cdots n_k)$ est définie et égale à $\overline{m}$, alors $\Psi^*\overline{n_1}\cdots\overline{n_k}\mapsto_\beta^*\overline{m}$. Toutefois, on ne peut pas conclure que $\Psi^*\overline{n_1}\cdots\overline{n_k}$ n'est pas normalisable quand $\psi(n_1,\cdots n_k)$ n'est pas définie, pour des raisons semblables à celles qui sont exposées dans l'exemple 5.1. L'idée est alors de forcer d'abord l'évaluation de $\Phi a_1\cdots a_k$, et, en cas de succès (i.e. de terminaison de la β-réduction), on réduit Ψ^*. Si $\Phi a_1\cdots a_k$ n'est pas normalisable, alors l'expression Ψ calculée ne sera pas normalisable non plus. Donc on définit plutôt Ψ par :

$$\Psi=\lambda a_1.\cdots\lambda a_k.((\Phi)a_1\cdots a_k)(\lambda b.b)\Psi^* a_1\cdots a_k.$$

Par le lemme précédent,

- $(\cdots((\Psi)\overline{n_1})\cdots)\overline{n_k} \mapsto_\beta^* (\overline{j})(\lambda b.b)\Psi * n_1\cdots n_k \mapsto_\beta^* \overline{\psi(n_1\cdots n_k)}$ si j est la plus petite valeur de m telle que $\chi(n_1\cdots n_k,m)=0$
- la β-réduction normale de

  $$(\cdots((\Psi)\overline{n_1})\cdots)\overline{n_k}$$

  ne termine pas s'il n'existe pas de m tel que $\chi(n_1\cdots n_k,m)=0$. Ceci à cause du fait que la réduction de la sous-λ-expression $(\Phi)a_1\cdots a_k$ ne termine pas.

■

On passe maintenant à la deuxième étape de notre preuve d'équivalence.

Théorème 5.9 *Les fonctions λ-représentables sont Turing-semi-calculables.*

Idée de preuve: Essentiellement, il faut se convaincre que l'on peut écrire un programme de machine de Turing pour faire l'évaluation à gauche d'une λ-expression. Soit F une λ-expression représentant une fonction partielle f. Considérons une machine M qui sur donnée $0^{n_1}10^{n_2}1...10^{n_k}$ commence par remplacer la donnée sur son ruban par $(...((F)\overline{n_1})...)\overline{n_k}$. L'alphabet de travail de M contiendra donc $\lambda, a, b, c, ..., x, y, z, ', ., (,)$. Pour faire la contraction, M n'a besoin que de la connaissance de F dans sa table de transition et de la possibilité de convertir 0^n en $\lambda x.\lambda y.\underbrace{(x)(x)...(x)}_{n}y$ ce qui est vraiment très simple. Ensuite, M doit faire la réduction normale de l'expression.

Pour faire cela, on va donner à M un certain nombre de rubans afin de se rendre la vie plus facile. M doit d'abord trouver le rédex le plus à gauche. Ceci est très simple, car M n'a qu'à trouver le $(\lambda.E$ le plus à gauche, puis de compter les (et) à l'aide d'un autre ruban lui servant de pile. M écrit sur un ruban le nom de la variable à substituer (disons x) et sur un autre ruban ce par quoi il faut substituer (disons E'). M trouve toute les occurrences libres de x dans E et les remplace explicitement par E'. Pour éviter tout conflit possible avec des variables existantes il est plus sage de renommer toutes les variables de E' par des variables nouvelles.

Si à un moment, M ne peut plus faire de β-contraction de l'expression elle convertit le résultat (un entier de Church $\lambda t.\lambda w.\underbrace{(t)(t)...(t)}_{m}w$, avec t et w deux noms quelconques de variables) en la valeur 0^m, puis s'arrête. ■

Théorème 5.10 *Les fonctions Turing-semi-calculables sont partielles récursives.*

Idée de preuve: Soit M une machine de Turing calculant une certaine fonction partielle f. Maintenant, on voudrait une fonction partielle récursive qui définit f. Supposons que M compte e états et t symboles de travail. Aux symboles de travail $s_1, \ldots, s_t$ on associe les nombres $1, \ldots, t$ et aux états $q_0, \ldots, q_{e-1}$ on associe les nombres $t+1, \ldots, t+e$. Si bien qu'une configuration

$\alpha q_i \beta$ de M est représentée par un nombre $\overline{\alpha}(t+i+1)\overline{\beta}$ écrit en base $t+e+1$, $\overline{s_{i_1} \dots s_{i_k}}$ étant le nombre $i_1 \dots i_k$.

Sur donnée $n_1, n_2, ..., n_k$, une fonction récursive va d'abord transformer $n_1, n_2, ..., n_k$ en un entier représentant la configuration initiale $q_0 2^{n_1} 1 2^{n_2} 1 ... 1 2^{n_k}$:

$$(t+1)\underbrace{222...2}_{n_1}1\underbrace{222...2}_{n_2}1...1\underbrace{222...2}_{n_k}.$$

On appelle cette transformation $\chi(\vec{n})$. Ensuite une fonction va transformer ce nombre en fonction des transitions de M (pour faire ceci il suffit de trouver la position de l'état puis de décider ce que ferait M dans la même situation, puis d'ajuster le nombre en conséquence).

La fonction $\varphi(m, \vec{n})$ aura pour résultat le codage de la configuration obtenue après m transitions à partir de la configuration initiale contenant $\vec{n}$. On aura une définition récursive de la forme suivante:

$$\varphi(m, \vec{n}) = \begin{cases} \chi(\vec{n}) & \text{si } m = 0 \\ \psi(\varphi(m-1, \vec{n}), m-1, \vec{n}) & \text{si } m > 0 \end{cases}$$

où $\psi(a, b, c)$ va convertir le nombre a représentant une certaine configuration en la configuration suivante selon les actions de M. Lorsque M s'arrête ou lorsque $a = 0$ (ce qui signifie que M s'est déjà arrêtée), alors $\psi(a, b, c) = 0$. Considérons maintenant la fonction suivante définie par minimisation partielle:

$$\delta(\vec{n}) = \min_m [\varphi(m, \vec{n}) = 0].$$

Si M s'arrête sur donnée $2^{n_1} 1 2^{n_2} 1 ... 1 2^{n_k}$ alors $\varphi(pred(\delta(\vec{n})), \vec{n})$ sera définie et aura pour valeur la dernière configuration avant que M ne s'arrête. Quand M ne s'arrête pas $\delta(\vec{n}) \uparrow$. On prendra donc

$$f = \nu(\varphi(pred(\delta(\vec{n})), \vec{n}))$$

où ν va faire la conversion inverse d'un nombre de la forme $\underbrace{22(t')2...2}_{n}$ en n ($t' > t$ code un état final de M). La fonction ν n'est pas définie pour les configurations du ruban qui ne codent pas un entier. La fonction f sera donc définie si et seulement si M s'arrête avec un résultat correct sur son ruban, et dans ce cas la valeur de f sera celle calculée par M. ■

Corollaire 5.11 *Soit f une fonction partielle de $N^k \to N$. Les propriétés suivantes sont équivalentes:*

- *f est partielle récursive*
- *f est Turing-semi-calculable*
- *f est λ-représentable.*

Corollaire 5.12 *Soit f une fonction totale de $N^k \to N$. Les propriétés suivantes sont équivalentes:*

- *f est récursive*
- *f est Turing-calculable*
- *f est λ-représentable (par une λ-expression toujours normalisable).*

Corollaire 5.13 *Soit E un ensemble d'entiers naturels et soit E' un ensemble de codages pour ces entiers (par exemple, n est codé comme une suite de n 0). Les deux propriétés suivantes sont équivalentes:*

- *E est récursivement énumerable*
- *E' est Turing-semi-calculable*

Preuve du lemme 5.13. Si E est récursivement énumerable, sa fonction caractéristique est récursive partielle et, par consequent, Turing-semi-calculable par une machine M. Soit M' une machine qui, sur une donnée (qui code un entier) n, simule M et accepte si et seulement si le résultat calculé par M est 1. Il est clair que $L(M') = E'$, donc E' est Turing-semi-calculable. Réciproquement, si E' est Turing-semi-calculable il existe une machine M' telle que $L(M') = E'$. Soit M une machine qui, sur une donnée n, simule M' sur (le codage de) n et donne comme résultat 1 si et seulement si M' accepte n. La fonction calculée par M', étant Turing-semi-calculable, est aussi récursive partielle et est la fonction partielle caractéristique de E; il en suit que E est récursivement énumerable.

De même, on a :

Corollaire 5.14 *Soit E un ensemble d'entiers naturels et soit E' un ensemble de codages pour ces entiers (par exemple, n est codé comme une suite de n 0). Les deux propriétés suivantes sont équivalentes:*

- *E est récursif*
- *E' est Turing-calculable*

Ce deux derniers corollaires montrent qu'en effet l'expression "récursivement énumerable" est synonyme de "Turing-semi-calculable", pour ce qui concerne les ensembles d'entiers. De même pour "récursif" et "Turing-calculable". En effet, dans la suite on généralisera cette synomimie à tout langage L et on dira souvent que L est récursif plutôt que dire que L est Turing-calculable; de même pour "L est récursivement énumerable" et "L est Turing-semi-calculable".

6 λ-calcul typé

Une première version du λ-calcul typé a été proposée dès 1940 par A. Church. On peut envisager plusieurs raisons (qui ne sont pas nécessairement celles de Church) à l'introduction des types en λ-calcul.

Tout d'abord, le λ-calcul est extrèmement riche puisqu'il permet, comme nous l'avons vu, de coder toutes les structures de données et tous les calculs. En fait, il est trop riche, car on peut aussi bien appliquer, par exemple, l'expression NOT à une expression qui n'est pas β-équivalent à un Booléen. Or on veut, et c'est aussi un principe général dans les langages de programmation, détecter les erreurs aussi tôt que possible. En particulier, on souhaite, par une interprétation abstraite des expressions détecter certaines erreurs sans pour autant évaluer les expressions. En λ-calcul typé NOT ne pourra être appliqué qu'à un Booléen.

Nous verrons aussi que la β-réduction (en fait, une notion de réduction plus générale que la β-réduction) termine toujours sur une λ-expression typé. Ce qui n'est évidemment pas le cas en λ-calcul pur. En programmation, comme dans les modèles de calcul correspondant, un programme est considéré comme correct si, entre autres, il termine toujours lorsque les données qui lui sont fournies sont du type voulu. En ce sens, le λ-calcul typé est un progrès vers un modèle dans lequel on ne pourrait écrire que des programmes corrects.

Cependant, ces avantages ont leur contrepartie: on ne pourra pas exprimer en λ-calcul typé les fonctions partielles, qui dans certains cas peuvent être utiles, même pour exprimer des fonctions totales. Par exemple, on n'aura pas d'opérateur de point fixe en λ-calcul typé. De toutes façons il est illusoire de tenter de définir un langage décidable correspondant exactement aux fonctions calculables. Insistons donc sur le fait que, contrairement aux autres modèles de calcul déjà présentés, le λ-calcul typé n'a pas le même pouvoir d'expression que les machines de Turing.

Enfin, bien que nous n'insistions pas sur cet aspect ici, le λ-calcul typé possède des liens profonds avec les sytèmes de preuve (en déduction naturelle). L'isomorphisme dit de "Curry-Howard" établit une correspondance entre expressions et preuves d'une part et types et formules d'autre part. Dans ce contexte, une expression t a le type α ssi t représente une preuve de la formule α. Par exemple, l'expression $\lambda x.x$ où x est de type α a pour type $\alpha \to \alpha$ (les fonctions de α dans α). À cette expression correspond une preuve de l'implication $\alpha \Rightarrow \alpha$.

Nous allons considérer ici un λ-calcul *simplement typé* qui est assez pauvre du point de vue de l'expressivité. Ce système a été étendu de bien des façons et nous évoquerons brièvement le sytème **T** de Gödel en fin de chapitre. Nous commencerons par donner la syntaxe des expressions du calcul, ainsi qu'un algorithme d'*inférence de types*. Nous verrons ensuite comment calculer: nous donnerons les règles de réduction de ce λ-calcul typé. Enfin, nous présenterons le résultat de *normalisation faible* qui énonce que toute expression du λ-calcul simplement typé admet une forme normale.

6.1 Syntaxe des expressions du λ-calcul simplement typé

6.1.1 Types

On suppose donné un alphabet $\mathcal{B}$ de *types atomiques* (aussi appelés *types de base*). Nous noterons ces types $A, B, \ldots$.

L'ensemble des types est alors défini par récurrence:

- tout type atomique est un type
- si σ et τ sont des types, alors $\sigma \to \tau$ et $\alpha \times \tau$ sont des types.

Etant donné un alphabet $X, Y, Z, \ldots$ de *variables de type*, l'ensemble des *schémas de type* est construit comme ci-dessus avec, de plus,

- toute variable de type est un schéma de type

6.1.2 Expressions

Les expressions du λ-calcul typé considéré ici sont les mêmes que ceux du λ-calcul pur, avec en, plus, les constructions suivantes:

- si E est une expression, alors $\pi_1 t$ et $\pi_2 t$ sont des expressions
- si E_1, E_2 sont des expressions, alors $< E_1, E_2 >$ est une expression.

Intuitivement, $< E_1, E_2 >$ est un couple d'expressions et π_1 et π_2 sont les projections sur les premier et deuxième éléments du couple respectivement. On aura alors les identités $\pi_1 < E_1, E_2 >= E_1$ et $\pi_2 < E_1, E_2 >= E_2$.

6.1.3 Expressions typées

La correspondance entre expressions et les types est assez intuitive: quand on forme une abstraction $\lambda x.E(x)$, on construit une fonction qui à t associe $E(t)$. Le domaine de cette fonction correspond au type de x et son image au type de E. Ainsi, si x a pour type σ et E a pour type τ, le type de $\lambda x.E$ est $\sigma \rightarrow \tau$: fonctions de σ vers τ.

De même, quand on forme un couple $< u, v >$, si σ est le type de u et τ est le type de v, alors $\sigma \times \tau$ est le type de $< u, v >$.

Ces règles de formation des expressions typés sont formalisées ci-dessous.

Un *contexte* C est un ensemble de couples, notés $x : \sigma$, formés d'une variable et d'un type et tels que chaque variable apparait au plus une fois dans C. Le contexte permet de typer les variables libres.

Les règles de typage sont données sous la forme de *règles de déduction*: les hypothèses (ou les propriétés déjà déduites) sont au dessus de la barre de fraction et la propriété qui s'en déduit se trouve au dessous de la barre de fraction. Les "propriétés" auxquelles nous nous interessons sont des *jugements* de la forme $C \vdash t : \sigma$ où C est un contexte, E est une expression et σ est un type. De tels jugements se lisent "dans le contexte C, E a le type σ.

Axiome	$\dfrac{}{C \cup \{x : \sigma\} \vdash x : \sigma}$
Affaiblissement	$\dfrac{C \vdash E : \sigma}{\{x : \tau\} \cup C \vdash E : \sigma}$ Si $x \notin C$
Application	$\dfrac{C \vdash E_1 : \sigma \rightarrow \tau \qquad C \vdash E_2 : \sigma}{C \vdash (E_1)E_2 : \tau}$
Abstraction	$\dfrac{\{x : \sigma\} \cup C \vdash E : \tau}{C \vdash \lambda x.E : \sigma \rightarrow \tau}$
Accouplement	$\dfrac{C \vdash E_1 : \sigma \qquad C \vdash E_2 : \tau}{C \vdash < E_1, E_2 >: \sigma \times \tau}$
Projection-g	$\dfrac{C \vdash E : \sigma \times \tau}{C \vdash \pi_1 E : \sigma}$
Projection-d	$\dfrac{C \vdash E : \sigma \times \tau}{C \vdash \pi_2 E : \tau}$

Une expression E est *typable* si l'on peut déduire le jugement $C \vdash E : \sigma$ pour un certain σ et un certain C. On dit alors que E *est de type* σ et que $E : \sigma$ est une expression typée.

Exemple 6.1 *$VRAI = \lambda x.\lambda y.x$ et $FAUX = \lambda x.\lambda y.y$ ont pour type(s) $\sigma \to (\sigma \to \sigma)$ où σ est un type quelconque. En effet, montrons, par exemple pour $VRAI$, comment obtenir ce résultat:*

$$\dfrac{\dfrac{\dfrac{}{x : \sigma, y : \sigma \vdash x : \sigma}\ \mathit{Axiome}}{x : \sigma \vdash \lambda y.x : \sigma \to \sigma}\ \mathit{Abstraction}}{\vdash \lambda x.\lambda y.x : \sigma \to (\sigma \to \sigma)}\ \mathit{Abstraction}$$

Exemple 6.2 *L'expression $(x)x$ n'est pas typable. En effet, si elle est typable, on doit pouvoir construire une preuve de la forme:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\underline{C_2 \vdash x : \sigma_1 \to \sigma}}{\vdots}}{C_3 \cup C_2 \vdash x : \sigma_1 \to \sigma}\ \mathit{Aff.} \qquad \dfrac{\dfrac{\underline{C_4 \vdash x : \sigma_1}}{\vdots}}{C_5 \cup C_4 \vdash x : \sigma_1}\ \mathit{Aff.}}{C_1 \vdash (x)x : \sigma}\ \mathit{Application}}{\dfrac{\vdots}{C \cup C_1 \vdash (x)x : \sigma}}\ \mathit{Aff.}$$

avec $C_1 = C_3 \cup C_2 = C_5 \cup C_4$.

On doit alors avoir $x : \sigma_1 \to \sigma \in C_2$ et $x : \sigma_1 \in C_4$. Mais cela n'est pas possible car $C_2 \subseteq C_1$ et $C_4 \subseteq C_1$: C_1 ne serait pas un contexte car il contiendrait deux occurrences de x.

Comme on le voit dans l'exemple ci-dessus la présence de la règle d'affaiblissement complique considérablement les preuves. Cette règle n'a en fait qu'une seule raison d'être: elle permet de lier plusieurs fois successivement une même variable. Par exemple, elle permet de typer $\lambda x.\lambda x.y$ qui ne peut pas l'être sans. Cependant, si l'on suppose que, chaque fois que c'est nécessaire, on utilise une α-conversion de sorte que chaque variable ne soit liée qu'une fois, alors la règle d'affaiblissement est inutile. Dans l'exemple, $\lambda x.\lambda x.y \equiv_\alpha \lambda z.\lambda x.y$ qui, lui, est typable sans l'affaiblissement. Nous ferons donc désormais cette hypothèse simplificatrice, éliminant la règle ennuyeuse de notre systéme.

6.1.4 Inférence de types

L'objectif est ici de présenter de façon générale un mécanisme permettant de décider si une expression est typable ou non. Ainsi, la preuve envisagée dans l'exemple 6.2 sera un cas particulier de l'algorithme que nous décrivons.

On dit que le schéma de type σ est un *type principal* de l'expression E si

- E a le type σ
- Pour tout (schéma de) type τ tel que E a le type τ, il existe une substitution S des variables de σ tel que $S(\sigma) = \tau$.

Exemple 6.3 *Dans l'exemple 6.1, $X \rightarrow (Y \rightarrow X)$ est un type principal de $VRAI$ et $X \rightarrow (Y \rightarrow Y)$ est un type principal de $FAUX$. En effet, quels que soient les types σ et τ, $VRAI : \sigma \rightarrow (\tau \rightarrow \sigma)$ est prouvable et $VRAI$ ne possède que les types de cette forme.*

Remarquons que, si l'on veut que $VRAI$ et $FAUX$ soient de même type (BOOL), il faut et il suffit de choisir un type de la forme $\sigma \rightarrow (\sigma \rightarrow \sigma)$: $X \rightarrow (X \rightarrow X)$ est un type principal pour à la fois *$VRAI$ et $FAUX$.*

Théorème 6.1 *On peut décider si un terme est typable. Tout terme typable possède un type principal.*

Idée de la preuve. On suppose, par α-conversion, que chaque variable de l'expression E est liée au plus une fois et n'apparait pas à la fois libre et liée.

On associe à chaque sous-expression E' de E une variable de type $X_{E'}$. On suppose bien entendu que ces variables sont distinctes pour des sous-expressions distinctes. L'expression E a le type σ si et seulement si on peut trouver une substitution S de ces variables telle que $S(X_E) = \sigma$ et telle que S satisfait les équations reflétant les constructions de E. Plus précisément, pour chaque sous-expression E' de E, on considère les équations suivantes:

$$\begin{array}{lcl}
E' = (U)V & \leadsto & X_U = X_V \rightarrow X_{E'} \\
E' = \lambda y.U & \leadsto & X_{E'} = X_y \rightarrow X_U \\
E' = y & \leadsto & X_{E'} = X_y \\
E' = < U, V > & \leadsto & X_{E'} = X_U \times X_V \\
E' = \pi_1 U & \leadsto & \exists Y. X_U = X_{E'} \times Y \\
E' = \pi_2 U & \leadsto & \exists Y. X_U = Y \times X_{E'}
\end{array}$$

Les équations reflètent en fait exactement les règles de typage.

Le système d'équations est résolu en utilisant un algorithme d'unification. Si le système n'a pas de solution, alors E n'est pas typable. Si, au contraire, le système d'équations a une forme résolue (unificateur le plus général) $X_E = T \wedge \ldots$, alors T est un type principal de E. ■

Exemple 6.4 *$(x)x$ n'est pas typable. En effet, si Y est le type de x et Z est le type de l'expression, on doit résoudre l'équation*

$$Y = Y \rightarrow Z$$

qui n'a pas de solution (finie).

Exemple 6.5 *$E = (\lambda x.x)\lambda x.x$ est typable. Pour montrer cela et calculer son type principal, commençons par appliquer une α-conversion: $E \equiv_\alpha (\lambda y.y)\lambda x.x$. On doit maintenant résoudre le système d'équations:*

$$\begin{array}{ll} & X_{\lambda y.y} = X_{\lambda x.x} \rightarrow X_E \qquad \wedge \quad X_{\lambda y.y} = X_y \rightarrow X_y \\ \wedge & X_{\lambda x.x} = X_x \rightarrow X_x \end{array}$$

On trouve comme forme résolue

$$\begin{array}{ll} & X_E = X_x \rightarrow X_x \wedge X_y = X_x \rightarrow X_x \wedge X_{\lambda y.y} = (X_x \rightarrow X_x) \rightarrow (X_x \rightarrow X_x) \\ \wedge & X_{\lambda x.x} = X_x \rightarrow X_x \end{array}$$

Le type principal de E est donc $X \rightarrow X$. (Une fonction d'un domaine dans lui-même).

6.2 Règles de réduction

En plus de la β-contraction ordinaire

$$(\lambda x.U)V \rightarrow [V/x]U$$

on dispose des règles de projection:

$$\begin{array}{l} \pi_1 < U, V > \rightarrow U \\ \pi_2 < U, V > \rightarrow V \end{array}$$

Ces règles de contraction, comme pour la β-réduction peuvent être appliquées à des sous-expressions d'une expression E. On note $\mapsto_S$ la relation de contraction par l'une des trois règles ci-dessus. Comme pour le λ-calcul pur, la relation de *réduction* associée est obtenue en effectuant une suite de contractions et/ou d'α-conversions. On note alors $E \mapsto_S^* E'$.

Tout d'abord, la relation de contraction *préserve les types*:

Lemme 6.2 *Si $E \rightarrow_S E'$ et $E : \sigma$, alors $E' : \sigma$.*

Preuve. En fait, c'est une conséquence des règles de formation des types. Supposons que, par α-conversion (qui ne change pas le type d'une expression) toutes les variables de E sont liées au plus une fois et n'apparaissent pas à la fois libres et liées dans E.

On procède alors par récurrence sur la taille de l'expression E.

- Si E est une variable, aucune contraction n'est possible et la propriété est vraie.

- si maintenant E n'est pas une variable et que le redex contracté n'est pas E lui-même, il suffit d'appliquer l'hypthèse de récurrence au redex.

- si $E = (\lambda x.U)V \mapsto_S [V/x]U$, alors soit $x : \tau$, $U : \tau'$ et $V : \tau''$, par les règles de formation des types on a $\lambda x.U : \tau \rightarrow \tau'$, donc $\tau'' = \tau$ et $\tau' = \sigma$. U ayant même type que E et V même type que x, $[V/x]U : \sigma$. (Ce qu'on peut à nouveau prouver par récurrence sur la taille de U, mais ceci est laissé au lecteur).

- si $E = \pi_1 < U, V >$ (ou $\pi_2 < U, V >$: le cas est symétrique) et $E' = U$, alors si $< U, V >: \tau \times \tau'$, par les règles de formation des types pour la projection, $E : \tau$ et, par les règles de formations des types pour la paire, $U : \tau$.

■

Nous admettrons le résultat suivant

Théorème 6.3 *Pour toutes λ-expressions bien typées E, E_1, E_2, si $E \mapsto_S^* E_1$ et $E \mapsto_S^* E_2$, alors il existe une λ-expression bien typée E_3 telle que $E_1 \mapsto_S^* E_3$ et $E_2 \mapsto_S^* E_3$.*

Ce résultat a un sens, grâce au lemme 6.2. Il exprime de plus que chaque expression typée possède au plus une forme normale (comme en λ-calcul pure). Mais ici, à la différence du λ-calcul pur, nous allons montrer que toute expression possède au moins une forme normale, qui est aussi unique par le résultat ci-dessus.

6.3 Théorème de normalisation faible

Nous allons montrer ici que tout terme a au moins une forme normale. C'est-à dire que, si l'on choisit bien dans quel ordre contracter les redex, la réduction se termine toujours.

En fait, on peut prouver un résultat plus fort (dit de "normalisation forte"): toute séquence de contractions se termine. Autrement dit, la "stratégie bien choisie" utilisée ici est en fait inutile. Cependant, nous ne prouvons pas ici le résultat de normalisation forte qui est beaucoup plus difficile.

Pour prouver ce résultat, nous allons donner une mesure (un entier) de la complexité des types et des termes, qui décroitra strictement chaque fois qu'un redex est contracté. Le problème en général est que la contraction d'un redex peut faire apparaître de nouveaux redex. Étant donnée une contraction $E \mapsto_S E'$, par contraction du redex R de E, on classe donc les redex de E' en plusieurs catégories:

1. les redex *inchangés* qui sont des redex présents dans E et dans E', mais pas dans R
2. les redex de E' qui sont des redex *modifiés* de E: ce sont les redex R' de E tels que R est un redex de R' distinct de R' lui-même.
3. les redex *dupliqués*: ce sont les redex de R présents (éventuellement dupliqués par substitution) dans E'.
4. les redex *créés* qu'on peut aussi classer suivant les cas:
   (a) projection créee par une projection: $R = \pi_1 << U, V >, W >$ et E contient la sous-expression $\pi_i R$. La contraction de R crée le redex $\pi_i < U, V >$. (On a aussi le cas symétrique en remplaçant π_1 par π_2).

(b) β-réduction créee par une projection: $R = \pi_1 < \lambda x.U, V >$ et E contient la sous-expression $(R)W$. La contraction de R crée alors le redex $(\lambda x.U)W$.

(c) projection créee par une β-réduction à l'extérieur du redex: $R = (\lambda x. < U, V >)W$ ou bien $R = (\lambda x.x) < U, V >$. et E contient la sous-expression $\pi_i R$, on creé alors le redex $\pi_i[W/x] < U, V >$ (ou bien $\pi_i < U, V >$).

(d) projection créee par une β-réduction à l'intérieur du redex: $R = (\lambda x.U) < V, W >$ et R contient une sous-expression $\pi_i x$ alors le remplacement de x par $< V, W >$ crée le redex $\pi_i < V, W >$ à l'intérieur de U.

(e) β-réduction créee par une β-réduction à l'extérieur de R: $R = (\lambda x.\lambda y.U)V$ (ou bien $R = (\lambda x.x)\lambda y.U$) et E contient une sous-expression $(R)W$. La contraction de R crée alors le redex $([V/x]\lambda y.U)W$ (ou bien $(\lambda y.U)W$).

(f) β-réduction créee par une β-réduction à l'intérieur du redex: $R = (\lambda x.U)\lambda y.V$ et U contient une sous-expression $(x)W$. La substitution de x par $\lambda y.V$ crée alors le redex $(\lambda y.V)W$.

Nous allons montrer que tous les redex crées sont plus petits (pour la mesure que nous allons donner) que celui qui les a fait naitre. On définit alors la *complexité* d'un type de façon suivante:

- $C(\sigma) = 1$ si σ est un type atomique
- $C(\sigma \times \tau) = C(\sigma \to \tau) = \max(C(\sigma), C(\tau)) + 1$

La complexité d'un redex est alors définie par:

- $C(\pi_i < U, V >) = C(\sigma \times \tau)$ si $U : \sigma$ et $V : \tau$
- $C((\lambda x.U)V) = C(\sigma \to \tau)$ si $V : \sigma$ et $U : \tau$.

Lemme 6.4 *Si R est un redex créé dans E par contraction de R_0, alors $C(R) < C(R_0)$.*

Preuve. Il s'agit essentiellement d'une vérification des 6 cas de création de redex. Nous résumons dans le tableau ci-dessous les complexités de R et

R_0 suivant les cas. Par convention nous posons $n = C(\sigma)$ où σ est le type de U, $m = C(\tau)$ oú τ est le type de V et $p = C(\zeta)$ où ζ est le type de W, $q = C(\xi)$ où ξ est le type de x, $r = C(\omega)$ où ω est le type de y.

cas	R_0	R	$C(R_0)$	$C(R)$
(a)	$\pi_1 << U, V >, W >$	$\pi_i < U, V >$	$\max(\max(n, m) + 1, p) + 1$	$\max(n, m) + 1$
(b)	$\pi_1 < \lambda x.U, V >$	$(\lambda x.U)W$	$\max(\max(q, n) + 1, m) + 1$	$\max(q, n) + 1$
$(c1)$	$(\lambda x. < U, V >)W$	$\pi_i[W/x] < U, V >$	$\max(q, \max(n, m) + 1) + 1$	$\max(n, m) + 1$
$(c2)$	$(\lambda x.x) < U, V >$	$\pi_i < U, V >$	$q + 1$	$\max(n, m) + 1$
(d)	$(\lambda x.U) < V, W >$	$\pi_i < V, W >$	$\max(q, n) + 1$	$\max(m, p) + 1$
$(e1)$	$(\lambda x.\lambda y.U)V$	$([V/x]\lambda y.U)W$	$\max(q, \max(r, n) + 1) + 1$	$\max(r, n) + 1$
$(e2)$	$(\lambda x.x)\lambda y.U$	$(\lambda y.U)V$	$q + 1$	$\max(r, n) + 1$
(f)	$(\lambda x.U)\lambda y.V$	$(\lambda y.V)W$	$\max(q, n) + 1$	$\max(r, m) + 1$

La décroissance est immédiate d'après le tableau, si l'on remarque que:

- Dans le cas (c2), $q = \max(n, m) + 1$ car x a le même type que $< U, V >$.
- Dans le cas (d), $q = \max(m, p) + 1$ car x a le même type que $< V, W >$.
- Dans le cas (e2), $q = \max(r, n) + 1$ car x a le même type que $\lambda y.U$.
- Dans le cas (f), $q = \max(r, m) + 1$ car x a le même type que $\lambda y.V$.

■

Théorème 6.5 (normalisation faible) *Tout terme du λ-calcul typé admet une forme normale.*

Preuve. Tout d'abord, la complexité d'une expression quelconque sera la complexité maximale des redex qu'elle contient.

Il nous suffit de bien choisir la stratégie de normalisation: nous réduirons toujours un redex de complexité maximale qui ne contient pas lui-même de redex de cette même complexité (c'est toujours possible de choisir un redex de cette façon).

Pour une expression E soit $N(E)$ le nombre de redex de E dont la complexité est $C(E)$. Montrons maintenant que, si $E \mapsto_S E'$ selon cette stratégie, alors,

ou bien $C(E) > C(E')$,

ou bien $C(E) = C(E')$ et $N(E) > N(E')$.

Là encore, envisageons les diverses possibilités de redex R' de complexité maximale dans E'.

- Si R' est un redex dupliqué, alors, par choix de la stratégie, il est de complexité strictement inférieure à celle de R. Donc $C(E') < C(E)$
- Par le lemme 6.4, si R est un redex créé, alors $C(R') < C(R) \leq C(E)$. Par conséquent, dans ce cas aussi $C(E) > C(E')$.

Dans les autres cas, le redex R' est un redex de complexité maximale de E et de E'. Par conséquent, si $C(E) \leq C(E')$, alors E et E' ont les mêmes redex de complexité maximale, excepté R qui a été contracté. Il en résulte que $C(E) = C(E')$ et $N(E) > N(E')$

Maintenant, raisonnons par l'absurde et supposons qu'il y ait une séquence de réduction infinie à partir de E: $E \mapsto_S E_1 \ldots \mapsto_S E_n \ldots$ respectant la stratégie de réduction imposée. La séquence $C(E), C(E_1), \ldots C(E_n), \ldots$ est une suite d'entiers natuels décroissante. Elle est donc constante pour $n \geq k$: $C(E_k) = C(E_{k+1}) = \ldots$. Dans ce cas, d'après ce que nous venons de voir, $N(E_k) > N(E_{k+1}) > \ldots$ ce qui produit une suite infinie décroissante d'entiers positif. C'est absurde. ■

6.4 Exercices

6.4.1

Supposant l'existence d'un unique type de base B, montrer que les entiers de Church sont typables et calculer leur(s) type(s). Même question pour les booléens.

6.4.2

Montrer que $\lambda x(x)x$ n'est pas typable mais que $\lambda x.(\lambda x.x)\lambda x.x$ l'est.

6.4.3

Donner deux termes du λ-calcul pur, qui sont β-équivalents et tels que l'un est typable et l'autre non.

6.4.4

Montrer que “typable” $\Rightarrow$ “normalisable” mais que la réciproque est fausse.

6.4.5

Donner des λ-termes A_1, A_2 vérifiant les propriétés suivantes.

1. $\forall x, y.((A_1)x)y \mapsto_\beta^* (y)x$
2. $\forall x, y.((A_2)x)y \mapsto_\beta^* (A_2)x$

Montrer qu'on peut choisir A_1 de façon à ce qu'il soit typable et donner son type principal. Montrer par contre que A_2 ne peut pas être typable.

6.4.6 Fonctions représentables sur les entiers

B désignera dans la suite un type de base.

1. Soit $E^0_{B\to B}$ (resp. E^0_B) l'ensemble des expressions en forme normale de type $B \to B$ (resp. B) construites sur $X_B = \{z_0, z_1, z_2, \ldots\}$ et $X_{B\to B} = \{y\}$. Montrer que $E^0_{B\to B} = \{y\} \cup \{\lambda z_i.(y)^k z_j \mid i, j, k \geq 0\}$ et $E^0_B = \{(y)^k z_i \mid i, k \geq 0\}$

2. Soient maintenant $E^1_{B\to B}$ (resp. E^1_B) l'ensemble des termes de type $B \to B$ (resp. B) construits sur $X_B = \{z_1, z_2, \ldots\}$, $X_{B\to B} = \{y\}$ et $X_{(B\to B)\to(B\to B)} = \{x\}$. Montrer que pour tout $e_1 \in E^1_{B\to B}$,

   - ou bien $e_1 \in E^0_{B\to B}$,
   - ou bien e_1 est de la forme $(x)^k e'_1$ où $e'_1 \in E^1_{B\to B}$ et e'_1 est ou bien y ou bien de la forme $\lambda z_i.e''_1$
   - ou bien e_1 est de la forme $\lambda z_i.((x)^k e'_1)e_2$ où $e'_1 \in E^1_{B\to B}$ est ou bien y ou bien de la forme $\lambda z_i.e''_1$ et $e_2 \in E^1_B$

   De même montrer que tout $e_2 \in E^1_B$ est ou bien dans E^0_B ou bien est de la forme $(y)^{k_1}((x)^{k_2} e_1)e'_2$ où $k_1, k_2 \geq 0$ et $e_1 \in E^1_{B\to B}$ et $e'_2 \in E^1_B$.

3. Montrer que pour tout $e_1 \in E^1_{B\to B}$ il existe un polynôme P_{e_1} tel que, pour tout entier n,

$$[\overline{n}/x]e_1 \mapsto_\beta^* \lambda z.(y)^k z \Rightarrow k \leq P_{e_1}(n)$$

et que, de même, pour tout $e_2 \in E^1_B$, il existe un polynôme P_{e_2} tel que, pour tout entier n,

$$[\overline{n}/x]e_2 \mapsto^*_\beta (y)^k z \Rightarrow k \leq P_{e_2}(n)$$

4. On suppose que le type des entiers de Church est $(B \to B) \to (B \to (B \to B)$ (cf. exercice 6.4.1). Déduire des questions précédentes que les seules fonctions sur les entiers de Church (i.e. expressions typables E telles que, pour toute entier n, il existe un entier m, tel que $(E)\overline{n}$ se réduit en $\overline{m}$) sont des polynômes.

6.4.7 Système T de Gödel

Le système T de Gödel est un λ-calcul typé enrichi dans lequel

Les types sont ceux du λ-calcul typé avec en outre deux types de base: Int et Bool.

Les expressions sont celles du λ-calcul typé avec les constructions supplémentaires:

- 0 est une expression de type Int
- SE est une expression de type Int si E est un expression de type Int
- $R(t, u, v)$ est une expression de type σ si u est une expression de type σ, t est une expression de type Int et v est une expression de type $\sigma \to (\mathsf{Int} \to \sigma)$. ($R$ est l'opérateur de *récursion primitive*).
- V, F sont des expressions de type Bool
- $IF(t, u, v)$ est une expression de type σ si t est de type Bool, u est de type σ et v est de type σ.

Les règles de calcul sont celles du λ-calcul typé plus les règles suivantes:

$$\begin{array}{rcl} R(0, u, v) & \mapsto & u \\ R(Sn, u, v) & \mapsto & ((v)R(n, u, v))n \\ IF(V, u, v) & \mapsto & u \\ IF(F, u, v) & \mapsto & v \end{array}$$

1. Définir une expression It du système T telle que

$$(((It)f)\overline{n})\overline{k} \mapsto^* (f)^n \overline{k}$$

2. Montrer comment définir la fonction d'Ackermann (cf. exercice 2.3.12) dans le système T.

7 Décidabilité et indécidabilité

Dans cette partie du cours on va s'intéresser à des questions du type :"Le langage L est-il décidable?", c'est à dire, existe-t-il un algorithme pour décider si un mot m appartient ou pas à L?

Comme on a déja vu, la notion de langage récursif est une formalisation de la notion de langage décidable. Donc, en effet, on se posera plutôt des question du type : "le langage L est-il récursif?", ce qui revient à dire : "L est-il Turing-calculable?"

Il existe des langages qui ne sont pas décidables mais qui sont "semi-décidables", c'est à dire tels qu'il existe un algorithme qui est capable de "reconnaître" ou "prouver" le fait qu'un mot m appartient au langage, mais qui n'est pas nécessairement capable de détecter le fait que m *n'appartient pas* au langage. On se posera aussi des questions du type : "L est-il semi-décidable ?". Puisque la notion de langage récursivement énumerable est une formalisation de celle de langage semi-décidable, en realité on se posera plutôt des question du type :"le langage L est-il récursivement énumerable?", ce qui revient à dire : "L est-il Turing-semi-calculable?"

En effet, tout ceci revient à se poser des questions de décidabilité pour des *problèmes*. Par exemple, soit le problème "Quelles sont les machines de Turing qui n'acceptent aucun mot?". Ce problème peut être formulée en termes d'appartenance à un langage sur $\{0, 1\}$ en encodant les machines de Turing par des mots constitués de 0 et de 1. C'est à dire que, étant donnée une machine M, décider si M n'accepte aucun mot revient à décider si le codage $< M >$ de M appartient ou pas au langage $L_\emptyset$ défini par

$$L_\emptyset = \{< M >\mid L(M) = \emptyset\}.$$

Comme on va le voir, ce langage n'est pas récursif, donc le problème de savoir, pour n'importe quelle machine M, si $L(M) = \emptyset$ est un problème indécidable.

En général, étant donné n'importe quel problème π dont les seules réponses possibles sont soit OUI soit NON (un tel problème est appellé *problème de décision*), on peut se donner un codage approprié qui permet d'identifier les instances de π pour lesquelles la réponse est OUI comme un langage L_π et formuler les questions de décidabilité par rapport à π comme des questions sur L_π.

7.1 propriétés des langages récursifs et r.é.

Etant donnée n'importe quelle machine de Turing qui s'arrête sur tous les mots, on peut toujours la réécrire avec deux états terminaux possibles nommés *oui* et *non*, pour dire que la machine accepte ou rejette le mot en donnée (*oui* est défini même si la machine ne s'arrête pas toujours). On va construire des preuves de certains théorèmes sur les propriétés des langages récursifs et r.é. en combinant des machines. À l'aide des états *oui* et *non* on va pouvoir combiner séquentiellement deux machines ou imbriquer une machine dans une autre.

Théorème 7.1 *Le complément d'un langage récursif est récursif.*

Preuve: Soit L un langage récursif et M une machine s'arrêtant sur tous les mots de Σ^* telle que $L(M) = L$. Construisont M' à partir de M en faisant accepter M' dans l'état *non* et rejeter dans l'état *oui*. Clairement M' s'arrête sur tous les mots de Σ^* (tout comme M) et $L(M') = \overline{L(M)}$. C'est-à-dire que M' est une machine qui s'arrête toujours telle que $L(M') = \overline{L}$. Donc $\overline{L}$ est récursif.
■

Théorème 7.2 *L'union de deux langages récursifs est récursive. L'union de deux langages r.é. est r.é..*

Preuve: Soient L_1 et L_2 deux langages récursifs, acceptés respectivement par les machines M_1 et M_2 (qui s'arrêtent toujours). Construisons M qui s'arrête toujours et qui accepte $L_1 \cup L_2$. Sur donnée ω, M exécute M_1 jusqu'a ce qu'elle arrive à l'état *oui* ou *non*. Si M_1 dit *oui* alors M accepte ω, sinon M exécute M_2. M accepte alors si et seulement si M_2 accepte. Clairement, M s'arrête toujours (car M_1 et M_2 s'arrêtent toujours) et $L(M) = L_1 \cup L_2$ puisque $\omega \in L_1 \Rightarrow M_1$ accepte $\omega \Rightarrow M$ accepte ω et de même pour L_2.

Pour des langages r.é. on ne peut pas combiner séquentiellement comme précédemment, car M_1 peut ne jamais s'arrêter. Il faut donc combiner "parallèlement" M_1 et M_2. On construit donc M en lui faisant exécuter alternativement une étape de M_1 et une étape de M_2 jusqu'à ce que l'une ou l'autre atteigne l'état *oui*. Dans ce cas M accepte. Sinon elle rejette si M_1 et M_2 atteignent l'état *non*.
■

Théorème 7.3 *Si un langage L et son complément $\overline{L}$ sont tous les deux r.é. alors L est récursif.*

Preuve: Soient M_1 une machine qui accepte L et M_2 une machine qui accepte $\overline{L}$. Comme pour le théorème précédent, on construit M en lui faisant exécuter alternativement une étape de M_1 et une étape de M_2 jusqu'à ce que l'une ou l'autre atteigne l'état *oui*. Dans ce cas M accepte si c'est M_1 qui accepte, et rejette si c'est M_2 qui accepte. M va nécessairement finir par s'arrêter que sa donnée soit ou non dans L.
■

Corollaire 7.4 *Un langage L est récursif si et seulement si L et son complément $\overline{L}$ sont tous les deux r.é..*

7.2 Codage des machines de Turing

On va maintenant se donner un codage des machines de Turing de façon à les décrire sous forme de mots sur un alphabet donné. Ces machines pourront être alors elles-mêmes des données pour d'autres machines. Par le théorème 3.10 l'on sait que tout langage pouvant être reconnu par une machine de Turing, peut l'être par une machine n'ayant que $\{0, 1, B\}$ comme alphabet de travail. On considère donc que les machines M de la forme

$$M = \{Q, \{0, 1\}, \{0, 1, B\}, \delta, q0, B, \{q1\}\}$$

avec $Q = \{q0, q1, ..., qn\}$ (n étant un nombre écrit en base 2) un ensemble fini d'états. Tous les langages r.é sur $\{0, 1\}$ peuvent être reconnus par une machine de cette forme. L'avantage de choisir une description aussi restrictive est que chacune de ces machines peut-être uniquement spécifiée par sa fonction δ.

Afin de simplifier la description de notre codage, nous choisissons de le construire sur l'alphabet $\{q, 0, 1, \text{“},\text{''}, \text{“}(\text{''}, \text{“})\text{''}, B, d, g, p, \text{“}\delta\text{''}, \text{“} =\text{''}, \text{“};\text{''}\}$ où d, g, p représentent respectivement $\rhd, \lhd, \bigtriangledown$. Remarquons que, à nouveau grâce au théorème 3.10, nous pourrons en fait supposer quand cela nous arrangera que l'alphabet est réduit à $\{0, 1, B\}$. Par exemple le mot

$$\delta(q0, 0) = (q0, 0, d); \delta(q0, 1) = (q0, 0, d); \delta(q0, B) = (q1, B, g); \delta(q1, 0) = (q1, B, p)$$

représente la machine de la figure 4.

Étant donnée une machine M, on note $< M >$ son codage. Pour simuler le comportement de M sur une donnée ω, on souhaite coder ω à la suite de $< M >$. Pour ce faire, on enrichit l'alphabet du symbole S qui sépare $< M >$ de ω. On obtient ainsi le codage $< M, \omega > = < M > S\omega$. À nouveau, $< M, \omega >$ pourra aussi être vu comme un mot de $\{0,1\}^*$.

7.3 Un langage non r.é.

Supposons que l'on énumère tous les mots sur $\{0,1\}$ dans l'ordre "canonique" (en commençant par les plus courts, puis en ordre lexicographique pour chaque longueur), et que l'on nomme ω_i le $i^{\text{ième}}$ mot dans l'énumération. Supposons maintenant que l'on énumère les machines, en nommant M_j la machine décrite par l'entier j interprété en binaire. Si j ne décrit pas de machine on dit que M_j est la machine telle que $L(M_j) = \emptyset$.

Supposons maintenat que l'on construise une table booléenne infinie où la case (i,j) contiendra un 1 si et seulement si $\omega_i \in L(M_j)$. On considère maintenant la diagonale de cette table. Les valeurs sur la diagonale correspondent à l'appartenance du mot ω_i au langage $L(M_i)$. On voit sur la figure 12 une portion de cette table hypothétique. On construit maintenant le langage L_d

$$
\begin{array}{cc|cccccc}
 & & \mathbf{j} \longrightarrow & & & & & \\
 & & 1 & 2 & 3 & 4 & 5 & 6 \ldots \\
\hline
 & 1 & \boxed{0} & 1 & 1 & 1 & 0 & 1 \\
\mathbf{i} & 2 & 1 & \boxed{1} & 0 & 1 & 0 & 0 \\
\downarrow & 3 & 1 & 0 & \boxed{1} & 1 & 1 & 1 \\
 & 4 & 0 & 0 & 1 & \boxed{1} & 0 & 1 \\
 & 5 & 1 & 1 & 0 & 1 & \boxed{0} & 1 \\
 & 6 & 0 & 1 & 1 & 1 & 1 & \boxed{0} \\
 & \ldots & & & & & &
\end{array}
$$

Figure 12: table hypothétique de $\omega_i \in L(M_j)$

à partir de cette table. On prend

$$L_d = \{\omega_i | \omega_i \notin L(M_i)\}.$$

On va maintenant montrer que L_d n'est pas r.é..

Théorème 7.5 *L_d n'est pas récursivement énumérable.*

Preuve: Supposons que L_d soit r.é., c'est-à-dire qu'il existe une machine M telle que $L(M) = L_d$. Puisque M est une machine de Turing, il existe un (même plusieurs) j_0 tel que $M = M_{j_0}$. Maintenant, la question que l'on se pose est: "est-ce que $w_{j_0} \in L(M_{j_0})$?".

$$\begin{aligned} \omega_{j_0} \in L(M_{j_0}) &\iff \omega_{j_0} \in L_d \\ &\iff \omega_{j_0} \notin L(M_{j_0}) \end{aligned}$$

Cette contradiction nous fait conclure qu'il n'existe pas de machine M telle que $L(M) = L_d$.
■

7.4 Machine de Turing universelle

Soit L_u le *langage universel*, c'est-à-dire $L_u = \{< M, \omega > | \omega \in L(M)\}$. On appel ce langage *universel* car la question $\omega \overset{?}{\in} L(M)$ est équivalente à $< M, \omega > \overset{?}{\in} L_u$.

Théorème 7.6 *L_u est récursivement énumérable.*

Preuve: On va d'abord décrire une machine de Turing $\mathcal{M}$ avec 3 rubans qui accèpte L_u. Le premier ruban de $\mathcal{M}$ contient sa donnée $< M, \omega >$. La tête de lecture sur ce ruban sera utilisée pour trouver les transitions de la forme $\delta(q_i, x_j)$ dans la description de M. Le second ruban de $\mathcal{M}$ correspondra au ruban de M (on y copiera ω au début). Enfin, le troisième ruban sera utilisé pour mémoriser les états de M. Le comportement de $\mathcal{M}$ sera:

1. Vérifier le format de la donnée sur le ruban 1 : si le ruban 1 ne contient pas la donnée d'une fonction de transition suivie d'un mot, $\mathcal{M}$ rejette immédiatement.

2. Copier ω sur le ruban 2. Initialiser le ruban 3 à $q0$ puis placer toutes les têtes en début de ruban.

3. Si le ruban 3 contient $q1$, ce qui signifie que M est dans l'état q_1, alors $\mathcal{M}$ s'arrête en acceptant.

4. Supposons que le symbole lu par $\mathcal{M}$ sur le ruban 2 est x_j et que q_i est le mot courant sur le ruban 3. Une recherche sur le ruban 1 du sous-mot $\delta(q_i, x_j)$ mènera ou bien à une transition de la forme $\delta(q_i, x_j) = (q_{i'}, x_{j'}, m)$, ou bien à aucune transition de ce type. Dans le premier cas, $\mathcal{M}$ va remplacer x_j par $x_{j'}$ sur le ruban 2, remplacera le mot q_i par $q_{i'}$ sur le ruban 3 et se déplacera sur le ruban 2 dans la direction donnée par m. Enfin, $\mathcal{M}$ retourne à l'instruction 3. Dans le second cas, $\mathcal{M}$ s'arrête en rejetant.

Il est clair que $\mathcal{M}$ accepte $< M, \omega >$ si et seulement si M accepte ω. Également, $\mathcal{M}$ va boucler à l'infini exactement quand M boucle à l'infini sur la donnée ω. Enfin, si M rejette ω en s'arrêtant, $\mathcal{M}$ en fera autant.
$\blacksquare$

La machine $\mathcal{M}$ décrite ci-dessus est suffisante pour obtenir la preuve du théorème mais on va se permettre de la raffiner un peu. Puisque l'on sait que toute machine à plusieurs rubans peut-être simulée par une machine à un seul ruban et que par le théorème 3.10 il existe une machine faisant le même travail avec seulement $\{0, 1, B\}$ comme alphabet de travail, alors il existe une machine du format prescrit à la section 7.2 faisant le même travail que $\mathcal{M}$. Appelons cette machine M_u. M_u est *la machine de Turing universelle.*

7.5 Réductions et problèmes indécidables

On découvre maintenant la notion de réduction entre différents problèmes indécidables. L'idée d'une réduction est de montrer qu'un langage L n'est pas r.é. en utilisant le fait qu'un autre langage L' ne l'est pas. Pour faire la preuve, on montre que s'il existait une machine M telle que $L = L(M)$ alors il existerait une machine M' telle que $L' = L(M')$ (on réduit donc L' à L). En sachant que L' n'est pas r.é., on en déduit par contraposition que M ne peut exister.

Le langage universel

Théorème 7.7 *L_u n'est pas récursif.*

Preuve: On va montrer que $\overline{L_u}$ n'est pas r.é. et on en conclura le théorème par le corollaire 7.4. Pour montrer que $\overline{L_u}$ n'est pas r.é. on réduit L_d à L_u, c'est à dire que l'on montre :

$$\overline{L_u} \text{ est r.é.} \Rightarrow L_d \text{ est r.é..}$$

Supposons qu'il existe une machine M pouvant reconnaître $\overline{L_u}$. On construit une machine M' qui accepte L_d. Sur donnée ω, M' trouve un i tel que $\omega = \omega_i$ (c' est une opération qu'une machine de Turing peut faire : voir TD). Avec ce i, M' construit la donnée $< M_i, \omega_i >$ et la donne à M. Si M accepte $< M_i, \omega_i >$ (d'où $< M_i, \omega_i > \in \overline{L_u}$), alors M' accepte ω_i sinon elle rejette (ou boucle). Il s'en suit que M' accepte ω_i si et seulement si M_i n'accepte pas ω_i, donc que $\omega_i \in L_d$. Mais, $\omega_i = \omega$, donc M' accepte exactement les mots de L_d. Puisque M' ne peut pas exister (par le théorème 7.5), M ne peut pas exister non-plus.
■

Le problème de l'arrêt

On va montrer que le problême de savoir si une machine de Turing M s'arrête ou pas sur un mot w est indécidable.

Théorème 7.8 *Le langage :*

$$L_{arr} = \{< M, w >: \ M \ s'arrete \ sur \ l'entree \ w\}$$

n'est pas récursif.

Preuve : on réduit L_u à L_{arr}, c'est à dire que l'on montre :

$$L_{arr} \text{ est récursif } \Rightarrow L_u \text{ est récursif..}$$

Supposons qu'il existe une machine M_{arr} qui reconnaît L_{arr} et qui s'arrête toujours. On pourrait alors construire une machine M' qui reconnaît L_u et qui s'arrête toujours : sur la donnée $< M, w >$, M' passe le contrôle à M_{arr} : si cette derniére s'arrête sur $< M, w >$ en "disant OUI", alors on simule M sur w et on accepte w si et seulement si M accepte w; si, par contre, M_{arr} s'arrête sur $< M, w >$ en "disant NON", alors on rejette. Puisque M' ne peut pas exister, M ne peut pas exister non-plus.
■

Indécidabilité de la logique du premier ordre

Théorème 7.9 *La satisfiabilité d'une formule du calcul des prédicats du premier ordre est indécidable.*

Ébauche de preuve
On réduit le problème de l'arrêt à celui de la satisfiabilité en calcul des]prédicats. On utilise 3 symboles de prédicats I, F, Δ sur les configurations (i.e. mots $\alpha q \beta$ où $\alpha, \beta \in \{0,1\}^*$). Etant donnés une machine M, de relation de transition δ, et un mot ω, I, F, Δ sont spécifiés par les formules:

$$
\begin{array}{rrcl}
 & I(q_0\omega) & & \\
\forall x, y. & F(xq_1y) & & \\
\forall x, y, z. & \Delta(x,y) & \leftarrow & \Delta(x,z) \wedge \Delta(z,y) \\
\forall x, y. & \Delta(xqay, xa'q'y) & & \text{pour tous } q, a \text{ tels que } \delta(q,a) = (q', a', \rhd) \\
\forall x, y. & \Delta(xbqay, xq'ba'y) & & \text{pour tous } q, a \text{ tels que } \delta(q,a) = (q', a', \lhd) \\
\forall x, y. & \Delta(xqay, xq'a'y) & & \text{pour tous } q, a \text{ tels que } \delta(q,a) = (q', a', \bigtriangledown)
\end{array}
$$

Soit Γ la conjonction des formules ci-dessus. La machine M ne s'arrête pas sur la donnée ω si et seulement si

$$\Gamma \wedge \neg \exists x, y. (F(y) \wedge I(x) \wedge \Delta(x,y))$$

est satisfiable. ■

De même,

Théorème 7.10 *La validité d'une formule du calcul des prédicats du premier ordre est indécidable.*

Le langage vide

On utilise à nouveau le théorème 7.7 pour prouver l'indécidabilité de plusieurs autres problèmes similaires.

Théorème 7.11 *Soit $L_\emptyset = \{< M > | L(M) = \emptyset\}$. Le complément de $L_\emptyset$, $\overline{L_\emptyset}$ est récursivement énumérable mais $L_\emptyset$ ne l'est pas.*

Preuve: D'abord pour montrer que $\overline{L_\emptyset}$ est r.é. il suffit de décrire une machine M' qui l'accepte. Considérons une machine non-déterministe M' qui sur donnée $< M >$ va choisir de façon non-déterministe un mot ω. Ensuite, M' simule le travail de M sur donnée ω. M' accepte $< M >$ si et seulement si M accepte ω. Or, si $< M > \in \overline{L_\emptyset}$, il existe au moins un choix de ω tel que $< M >$ accepte ω; par consequent, si $< M > \in \overline{L_\emptyset}$, M' accepte $< M >$. Réciproquement, si $< M > \notin \overline{L_\emptyset}$, il n'existe aucun mot ω accepté par M, donc la machine non deterministe M' n'accepte pas $< M >$. Par suite, $L(M') = L_\emptyset$; par l'existance de M', $\overline{L_\emptyset}$ est r.é..

On voudrait maintenant montrer que $L_\emptyset$ n'est pas r.é., et, par conséquent, pas récursif.

On commence par observer le fait suivant : il existe une machine de Turing A telle que, pour toute entrée $< M, \omega >$, A engendre le codage $< T_{M,\omega} >$ d'une machine $T_{M,\omega}$ telle que :

$$L(T_{M,\omega}) = \emptyset \text{ si M n'accepte pas } \omega$$

$$L(T_{M,\omega}) \neq \emptyset \text{ si M accepte } \omega.$$

I suffit en effet d'écrire tout d'abord $< M, \omega >$ puis de simuler le comportement de M, en complétant la table de transitions de façon à ce que la machine "boucle" lorsqu'aucune transition de M n'était possible, sans être dans un état acceptant.

Maintenant, supposons par l'absurde que $L_\emptyset$ soit r.é.; il existe alors M' qui accepte tous les $< M >$ tels que $L(M) = \emptyset$. On va utiliser ce fait pour obtenir une machine M'' pour $\overline{L_u}$.

Sur la donnée $< M, \omega >$, M'' simule la machine A et engendre le codage $< T_{M,\omega} >$. Puis M'' exécute M' sur la donnée $< T_{M,\omega} >$ et accepte $< M, \omega >$ si et seulement si M' accepte $< T_{M,\omega} >$. Puisque par construction M' accepte $< T_{M,\omega} >$ si et seulement si $L(T_{M,\omega}) = \emptyset$, c'est à dire si et seulement si M n'accepte pas ω, par conséquent M'' accepte $< M, \omega >$ si et seulement si $< M, \omega > \in \overline{L_u}$.

Comme $\overline{L_u}$ n'est pas r.é., $\overline{L_\emptyset}$ non-plus.

■

7.6 Problème de correspondance de Post

Definition 7.1 *Soit Σ un alphabet fini et $A = (a_1, \ldots, a_n)$, $B = (b_1, \ldots, b_n)$ deux séquences finies de mots de Σ^*. On dit qu'une suite d'entiers $i_1, \ldots, i_k \in$*

$\{1, \ldots, n\}$, $k \geq 1$, est une solution au problème de correspondance de Post pour A, B si

$$a_{i_1} \ldots a_{i_k} = b_{i_1} \ldots b_{i_k}.$$

Théorème 7.12 *Le langage*

$$L_{Post} = \{(A, B) \mid \textit{ il existe une solution au problème de Post pour } A, B\}$$

n'est pas récursif.

Preuve
L'objectif est de coder le problème de l'arrêt d'une machine de Turing en un problème de correspondance de Post. Soit

$$M = (Q, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B, \{q_1\})$$

avec $Q = \{q_0, \ldots, q_n\}$ et $\omega \in \{0, 1\}^*$. On suppose, sans perdre de généralité que M commence par marquer son origine (au prix éventuel du déplacement de ω vers la droite). On construit d'abord une machine M' qui s'arrête sur un mot ω si et seulement si M s'arrête sur ω et qui possède la propriété supplémentaire suivante: si M' s'arrête, alors le ruban est vide et M' a effectué un nombre impair de transitions. Pour construire M' on rajoute 2 états q_f^1 et q_f^2 qui seront les nouveaux états finaux, un état q_t et les transitions qui

- à partir de q_1 déplacent la tête de lecture à l'extrémité gauche du ruban, en restant dans l'état q_1. Arrivé à cette extrémité, on passe dans l'état q_t
- à partir de q_t et de l'extrémité gauche du ruban, se déplacent jusqu'à l'extrémité droite en effaçant tout le contenu du ruban, puis on passe dans l'état q_f^1
- de q_f^1 on passe dans q_f^2 sans aucune action.

On considère alors l'alphabet

$$\Sigma = Q \cup Q' \cup \{0, 1, B, 0', 1', B', [,], *, *'\}$$

à $2 \times |Q| + 10$ éléments où $Q' = \{q'_0, \ldots, q'_n, q'_t, q'^1_f, q'^2_f\}$. Σ sera notre alphabet du problème de correspondance de Post. Il nous reste à définir A et B.

A	B	A	B
0	$0'$	1	$1'$
$0'$	0	$1'$	1
$*'$	$*$	$*$	$*'$
g_i	d'_i	g'_i	d_i
$[q_0 w*$	$[$	$]$	$*' q^1_f]$
$]$	$*' q^2_f]$		

où

- à toute transition $\delta(q_k, a) = (q_j, b, m)$ on associe les paires (g_i, d_i) comme suit:
  - si $m = \rhd$ et $q_k \notin \{q_t, q^1_f\}$, alors g_i est le mot $q_k a$ et d_i est le mot $b q_j$
  - si $m = \lhd$ et $q_k \notin \{q_t, q^1_f\}$, alors on associe à la règle de transition les deux paires $(g_i, d_i) = (0 q_k a, q_j 0 b)$ et $(1 q_k a, q_j 1 b)$
  - si $m = \bigtriangledown$ et $q_k \notin \{q_t, q^1_f\}$, alors g_i est le mot $q_k a$ et d_i est le mot $q_j b$
  - si $q_k = q_t$, alors $b = B$ et $m = \rhd$. Alors $g_i = q_t a$, $d_i = q_j$.
  - si $q_k = q^1_f$ et $q_j = q^2_f$, alors $g_i = q^1_f$ et $d_i = q^2_f$.
- g'_i, d'_i sont les mots obtenus en primant chacune des lettres de g_i et d_i respectivement.

M accepte ω si et seulement si

$$u_1 = q_0 \omega B^k \vdash_M u_2 \ldots \vdash_M u_m = q_f$$

où $q_f \in \{q^1_f, q^2_f\}$ et m est impair.

Montrons alors que le problème de Post possède une solution. Pour cela, il suffit de remarquer les deux décompositions du mot w suivant:

$$\begin{array}{rllllll} w = & [u_1* & u'_2*' & u_3* & \cdots &] & \\ = & [& u_1* & u'_2*' & \cdots & *'u_m] & \end{array}$$

Les mots en dessous les uns des autres étant en correspondance. En effet, u_i et u_{i+1} s'écrivent respectivement $u_i = v_i g_i w_i$ et $u_{i+1} = v_i d_i w_i$. Par définition des suites A et B, u_i et u'_{i+1} d'une part et u'_i et u_{i+1} d'autre part sont donc bien en correspondance.

Réciproquement, supposons que le problème de correspondance de Post ait une solution $w = a_{i_1} \dots a_{i_k} = b_{i_1} \dots b_{i_k}$. Alors a_{i_1} ne peut pas commencer par un symbole distinct de [,], car son correspondant dans B commencerait par un symbole primé si a_{i_1} commence par un symbole non primé et inversement b_{i_1} commencerait par un symbole non primé si a_{i_1} commençait par un symbole primé. De plus, a_{i_1} ne peut être], car b_{i_1} ne pourrait alors être que $*' q_f]$. Par conséquent, w ne peut commencer que par $[\omega *$. Cela prouve que $b_{i_1} = [$. De même, $a_{i_k} =]$ et $b_{i_k} = *' q_f]$ avec $q_f \in \{q_f^1, q_f^2\}$. Par récurrence sur la longueur de w', préfixe de w, on obtient que $w = [u_1 * u'_2 *' \cdots *' u_m]$ avec, pour tout i, $u_i = v_i g_i w_i$ et $u_{i+1} = v_i d_i w_i$ pour un certain couple (g_i, d_i). ■

7.7 Théorèmes de Rice

On considère maintenant des *propriétés* des langages récursivement énumérables sur $\{0,1\}$. Une propriété $\mathcal{L}$ est un sous-ensemble de l'ensemble des langages récursivement énumérables sur $\{0,1\}$. Par exemple, l'ensemble $\mathcal{L} = \{L | L \text{ est récursif}\}$ est une propriété des langages récursivement énumérables. On dit d'un langage r.é. L qu'il a la propriété $\mathcal{L}$ si $L \in \mathcal{L}$. Une propriété est *banale* si $\mathcal{L} = \emptyset$ ou que $\mathcal{L} = \{\text{langages r.é. sur } \{0,1\}\}$. On associe à $\mathcal{L}$ le langage $L_{\mathcal{L}} = \{< M > | L(M) \in \mathcal{L}\}$, le langage des machines qui reconnaissent des langages dans $\mathcal{L}$. Une propriété est dite *décidable* (resp. *semi-décidable*) si $L_{\mathcal{L}}$ est récursif (resp. r.e.).

Le théorème suivant nous donne un critère très général mais très simple permettant de décider si une propriété des langages récursivement énumerables est décidable.

Théorème 7.13 (Rice 1) *Toute propriété non banale $\mathcal{L}$ des langages r.é. sur $\{0,1\}$ est indécidable.*

Preuve: Soit $\mathcal{L}$ une propriété non banale des langages r.é. sur $\{0,1\}$. Sans perdre de généralité, on suppose que $\emptyset$ n'a pas la propriété $\mathcal{L}$, (sinon on peut refaire le même raisonnement pour la propriété $\overline{\mathcal{L}}$, qui est encore non banale).

Puisque $\mathcal{L}$ n'est pas banale, il existe un langage L r.é. avec la propriété $\mathcal{L}$. On observe le fait suivant. Il existe une machine de Turing B telle que, pour toute entrée $< M, \omega >$, B engendre le codage $< T'_{M,\omega} >$ d'une machine $T'_{M,\omega}$ telle que :

$$L(T'_{M,\omega}) = \emptyset \; si \; M \; n'accepte \; pas \; \omega$$

$$L(T'_{M,\omega}) = L \; si \; M \; accepte \; \omega.$$

On peut remarquer que L a la propriété $\mathcal{L}$ (par construction) mais que $\emptyset$ n'a pas la propriété $\mathcal{L}$. Donc la machine $T'_{M,\omega}$ reconnait un langage avec la proprieté $\mathcal{L}$ si et seulement si M accepte ω.

Supposons par l'absurde que $\mathcal{L}$ soit décidable, donc qu'il existe une machine $M_{\mathcal{L}}$ qui décide si un mot $< M >$ appartient ou pas à $\mathcal{L}$. On montre alors que l'on peut construire une machine $\overline{U}$ qui décide L_u (ce qui est absurde). La machine $\overline{U}$ que l'on va construire utilise $M_{\mathcal{L}}$ et B.

Sur la donnée $< M, \omega >$, $\overline{U}$ commence par simuler B et engendre le codage $< T'_{M,\omega} >$ de la machine $T'_{M,\omega}$. Puis elle appelle la machine $M_{\mathcal{L}}$ sur l'éntree $T'_{M,\omega}$. Si $M_{\mathcal{L}}$ réponde OUI, ceci veut dire que $L(T'_{M,\omega}) = L$, donc que M accepte ω. Si $M_{\mathcal{L}}$ réponde NON, ceci veut dire que $L(T'_{M,\omega}) = \emptyset$, donc que M n'accepte pas ω. En imposant que $\overline{U}$ accepte quand $M_{\mathcal{L}}$ répond OUI et rejette quand $M_{\mathcal{L}}$ répond NON, on obtient que $\overline{U}$ décide L_u.
■

Ce théorème très puissant nous permet de dire immédiatement qu'un grand nombre de propriétés sont indécidables.

Exemple 7.1 *Les propriétés suivantes des langages récursivement énumérables sont indécidables:*

- *L est vide.*
- *L est fini.*
- *L est régulier.*
- *L est algébrique.*

Par contre, la condition sous laquelle un langage $L_{\mathcal{L}}$ est récursivement énumérable est beaucoup plus complexe. Nous donnons le thóréme suivant sans le prouver.

Théorème 7.14 (Rice 2) *Soit une propriété non banale $\mathcal{L}$ des langages r.é. sur $\{0,1\}$. $L_{\mathcal{L}}$ est r.é. si et seulement si*

1. *Si L est dans $\mathcal{L}$ et que $L \subseteq L'$ pour un certain L' r.é., alors L' est aussi dans $\mathcal{L}$.*
2. *Si L est un langage infini dans $\mathcal{L}$, alors il existe un langage fini L' tel que $L' \subset L$ et $L' \in \mathcal{L}$.*
3. *L'ensemble des langages finis de $\mathcal{L}$ est* énumérable, *c'est-à-dire qu'il existe une machine de Turing qui engendre sur son ruban de travail le mot (éventuellement infini)*

   $$code_1\#code_2\#\ldots$$

   où chaque $code_i$ est un codage du $i^{\text{ième}}$ ensemble fini de $\mathcal{L}$.

Corollaire 7.15 *Les propriétés suivantes ne sont pas r.é.:*

- *L est vide.*
- *$L = \Sigma^*$.*
- *L est récursif.*
- *L n'est pas récursif.*
- *L est un singleton.*
- *L est régulier.*
- *$L - L_u$ n'est pas vide.*

Preuve: Dans tous les cas la condition 1) n'est pas respectée, sauf le second où la condition 2) n'est pas respectée et le dernier où la condition 3) n'est pas respectée.
■

Corollaire 7.16 *Les propriétés suivantes sont r.é.:*

- *L n'est pas vide.*

- *L contient au moins k éléments.*
- *le mot ω est dans L.*
- *$L \cap L_u$ n'est pas vide.*

(Preuve en TD).

7.8 Langages algébriques et indécidabilité

On peut montrer un grand nombre de résultats sur les grammaires génératrices de langages algébriques. Plusieurs des preuves de ces énoncés seront présentées en TD.

Théorème 7.17 *Soient G_1, G_2 deux grammaires génératrices de langages algébriques L_1, L_2. Il est indécidable de savoir si $L_1 \cap L_2 = \emptyset$.*

Théorème 7.18 *Soit G la grammaire génératrice d'un langage algébrique L. Il est indécidable de savoir si G est ambigüe.*

Théorème 7.19 *Soit G la grammaire génératrice d'un langage algébrique L. Il est indécidable de savoir si $L = \Sigma^*$.*

Théorème 7.20 *Soient G_1, G_2 deux grammaires génératrices de langages algébriques L_1, L_2. Il est indécidable de savoir si $L_1 = L_2$.*

7.9 Une hiérarchie de problèmes indécidables

Il est naturel de se poser la question:"si l'on savait décider quelles machines accepte le langage vide est-ce que l'on pourrait reconnaître n'importe quel langage?". (Est-ce que tous les langages indécidables deviendraient du même coup décidables?) Afin de poser cette question correctement, on va devoir définir précisément ce que l'on entend par là.

machines de Turing avec Oracle

On considère des machines de Turing avec oracles. Ce sont des machines de Turing avec trois état spéciaux: $q_?, q_{oui}, q_{non}$. À tout moment, une telle machine a le droit de passer dans l'état $q_?$ ou elle "interroge un oracle" afin de savoir si le mot situé à droite de sa tête de lecture appartient ou pas à un certain langage A. La machine se retrouve alors dans l'état q_{oui} si elle a fourni un mot dans A et se retrouve dans l'état q_{non} autrement. Elle continue ensuite son calcul. On note M^A une machine M avec un oracle pour le langage A et $L(M^A)$ le langage accepté par cette machine.

Remarquons que si A est un langage récursif, la machine M pourrait aussi bien faire le travail sans utiliser un oracle pour A puisqu'il existe une machine décidant l'appartenance à A. Par contre, si A n'est pas récursif il se peut que le langage reconnu par M^A ne soit pas un langage r.é. (par exemple M^A pourrait reconnaître A).

Un langage L est *récursivement énumérable par rapport à* A s'il existe une machine M^A telle que $L(M^A) = L$. Un langage L est *récursif par rapport à* A s'il existe une machine M^A qui s'arrête toujours telle que $L(M^A) = L$. Deux langages oracles sont *équivalents* si ils sont mutuellement récursifs l'un par rapport à l'autre.

hiérarchie

On peut maintenant reformuler notre question initiale en se demandant quels sont les langages récursifs par rapport à $A = L_\emptyset$. De toute évidence, tous les langages ne peuvent être reconnus par rapport à $L_\emptyset$ puisque qu'il existe un nombre non dénombrable de langages et seulement un nombre dénombrable de machines. Considérons donc le langage

$$S_1 = L_\emptyset = \{< M > | L(M) = \emptyset\}$$

qui n'est pas un langage r.é.. Considérons maintenant le problème de savoir si le langage engendré par une machine M par rapport à S_1 est vide. Ce problème n'est pas r.é. par rapport à S_1, mais nous ne le prouvons pas ici. De la même façon on définit une hiérarchie de langages "de plus en plus" indécidables par

$$S_i = \left\{< M^{S_{i-1}} > | L(M^{S_{i-1}}) = \emptyset\right\}.$$

On va pouvoir maintenant mesurer le degré d'indécidabilité de certains problèmes en montrant qu'ils sont équivalents à l'un ou l'autre des S_i (mais, bien sur, puisqu'il n'y a qu'un nombre dénombrable de S_i, les langages équivalents à un des S_i ne constituent qu'une infime partie de tous les langages possibles).

Exemple 7.2 *L_u est équivalent à S_1.*

Exemple 7.3 *$L_{\Sigma^*} = \{< M > | L(M) = \Sigma^*\}$ est équivalent à S_2.*

Exemple 7.4 *$L_{reg} = \{< M > | L(M)$ est régulier $\}$ est équivalent à S_3.*

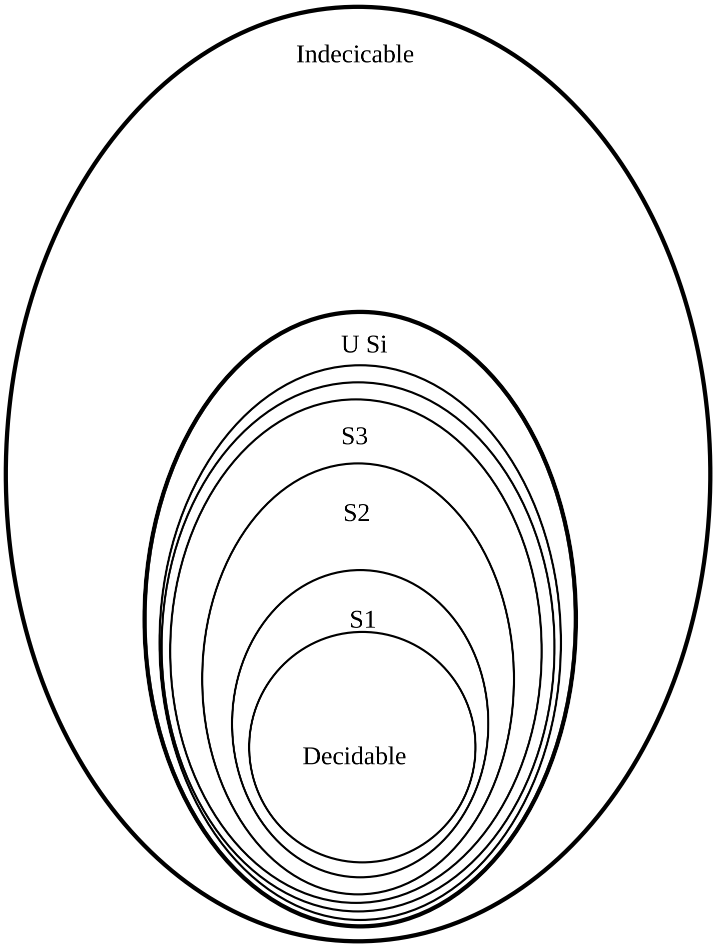

Figure 13: Hiérarchie de langages indécidable

PARTIE II : COMPLEXITE

8 Mesures et classes de complexité

Dans la première partie de ce cours, on a considéré plusieurs formalisations de la notion de fonction calculable et on a étudié des questions du type : le problème P est-il décidable ou pas ? Mais on ne s'est pas interessé à des questions de complexité de l'algorithme qui éventuellement décide le problème; c'est bien ce type de questions qui fait l'objet de la deuxième partie du cours.

Il faut d'abord remarquer que des trois formalisations équivalentes de la notion de fonction calculable étudiés - fonction récursive, λ-calcul et machine de Turing - c'est la dernière qui se prête le mieux à une étude du point de vue de la complexité, en temps ou en espace memoire, des algorithmes. C'est donc exclusivement ce troisième modèle de calcul que nous considérerons ici.

Il faut aussi rappeler que, modulo un codage approprié, on peut identifier les instances d'un problème de décision π pour lesquels la réponse est OUI comme un langage L_π. Donc, on peut parler de la complexité du problème π en parlant de la complexité de la machine de Turing qui accepte L_π. Sans perte de généralité, on se limitera aux langages sur $\{0,1\}^*$.

On va considérer principalement deux mesures de complexité, l'une liée à l'espace et l'autre liée au temps nécessaires à toute procédure de décision pouvant identifier les éléments d'un langage. Il existe de nombreuses autres mesures de complexité, comme par exemple le nombre de changement de direction de la tête d'une machine de Turing pendant son calcul, mais nous nous contenterons des deux plus importantes.

8.1 mesures de complexité

Considérons la machine de Turing M de la figure 14. M est une machine dite "off-line". Elle est équipée d'un ruban en lecture seulement avec sa donnée délimitée par "&" et "$" et de k rubans de travail où M peut lire et écrire.

complexité d'espace

Si pour tout mot en donnée de longueur n, la machine M regarde *au maximum* $f(n)$ cases sur chacun de ses rubans de travail on dira que M est une machine de *complexité d'espace* $f(n)$ (on dira aussi que M est *bornée en espace* par $f(n)$). Notons que si, pour au moins un mot w, il n'y a pas de borne au nombre de cases utilisées pour le calcul de M sur w, alors la complexité en espace de M n'est pas définie. On dit également d'un langage L qu'il est de complexité d'espace $f(n)$ s'il existe une machine M de complexité d'espace $f(n)$ telle que $L(M) = L$. On remarquera que la complexité d'espace de M est définie de façon telle qu'il s'agit de la complexité *dans le pire des cas*.

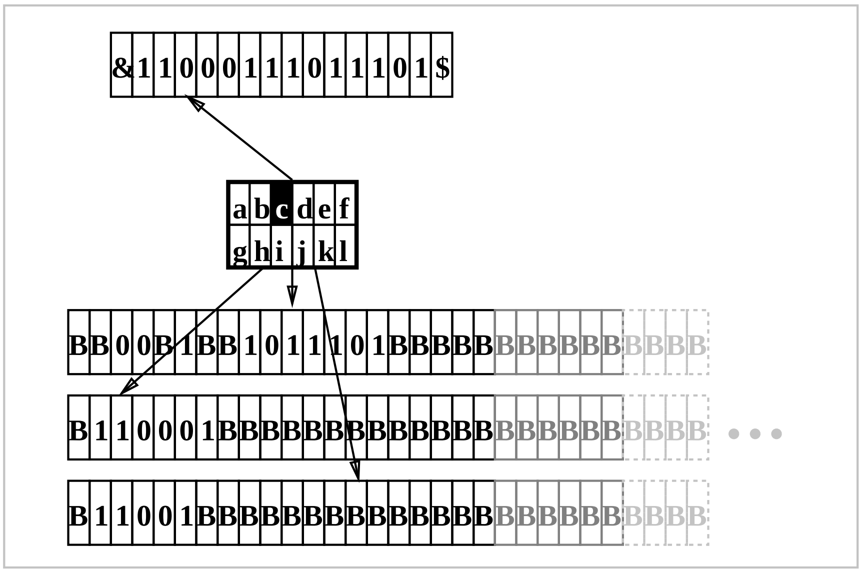

Figure 14: machine de Turing “off-line”

La raison pour laquelle on considère la complexité en espace sur une machine “off-line” est que sur une machine standard il serait impossible de définir des langages ayant une complexité inférieure à linéaire. Sur une machine du type que l'on a choisi, il nous sera possible de parler de langages de complexité d'espace $\log(n)$ par exemple. On supposera que la complexité en espace d'une machine est en tous cas ≥ 1; donc si on dit que L a une complexité en espace $f(n)$ ce que l'on comprend est que sa complexité en espace est égale à $max(1, \lceil f(n) \rceil)$. Pour des machines “off line” (et pour une donnée fixée), une configuration sera décrite par d'une part la position de la tête de lecture sur la donnée, d'autre part le contenu des k rubans de travail (au moyen d'une configuration habituelle).

complexité en temps

Considérons la machine de Turing M de la figure 15. M est une machine dite “bi-directionelle”. Elle est équipée de k rubans de travail bi-directionels, sa donnée étant sur son premier ruban. Pour ce type de machine, la fonction de transition δ a pour domaine $Q \times \Gamma^k$ (s'il y a k rubans) et donne comme résultat un élément de $Q \times \Gamma^k \times \{\rhd, \lhd, \bigtriangledown\}^k$. Les mouvements des têtes de lecture sont donc supposés “simultanés”.

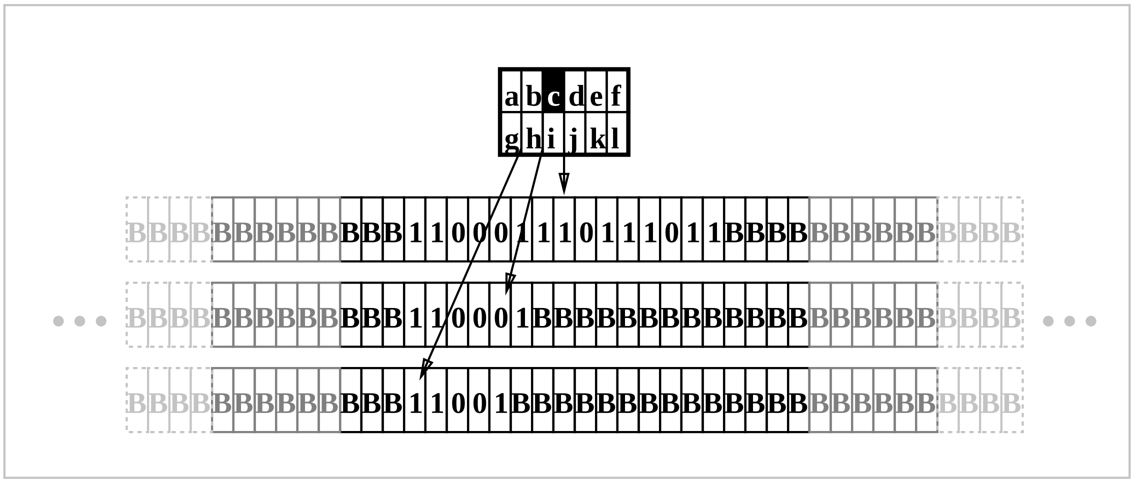

Figure 15: Machine de Turingà rubans infinis bidirectionels

Si pour tout mot en donnée de longueur n, la machine M fait *au maximum* $f(n)$ transitions avant de s'arrêter on dira que M est une machine de *complexité en temps* $f(n)$. (On dit aussi que M est *bornée en temps* par $f(n)$). Notons que si la machine ne s'arrête pas toujours, alors la complexité en temps n'est pas définie. On dit également d'un langage L qu'il est de complexité en temps $f(n)$ s'il existe une machine M de complexité de temps $f(n)$ telle que $L(M) = L$. Comme pour l'espace, on remarquera que la complexité en temps de M est définie de façon telle qu'il s'agit de la complexité *dans le pire des cas*. On supposera que la complexité en temps d'une machine est en tous cas $\geq n + 1$, car cela est le temps nécessaire à lire la donnée et réperer le premier blanc; ceci revient à dire que si l'on dit d'un langage L qu'il est est : complexité de temps $= max(n + 1, \lceil f(n) \rceil)$.

Le choix de deux différents modèles pour la complexité en espace et en temps permettra de faciliter l'expression de certaines preuves. Certaines variations auraient été possibles sur le modèle mais pas nécessairement toutes.

non déterminisme

Une machine non déterministe est de complexité en espace (ou en temps) $f(n)$ si pour toutes les données de taille n et tous les choix non-déterministes possibles elle utilise au plus $f(n)$ cases de ses rubans (prend au plus $f(n)$ étapes avant de s'arrêter). A nouveau, il s'agit d'une complexité dans le pire des cas, pour deux raisons :

- on considère *tous* les mots de taille n pour calculer la borne $f(n)$ (donc les configurations les plus coûteuses sont prises en compte)
- on considère tous les suites de choix possibles (donc les suites les plus coûteuses sont prises en compte)

Une suite de choix pour une machine non déterministe M, à partir d'une entrée w, détermine une suite de descriptions instantanées, qui ne répresente rien d'autre qu'un calcul possible de M pour w. Toutes les suites possibles de descriptions instantanées de M pour w constituent les branches d'un arbre $T(M, w)$ où, pour tout sommet, le nombre des fils est fini. La définition ci-dessus nous dit que M est de complexité en espace $f(n)$ si pour n'importe quel mot w de taille n, toute branche β de $T(M, w)$ est telle que les descriptions de la branche ne font pas intervenir plus que $f(n)$ cases distinctes. De même, M est de complexité en temps $f(n)$ si pour n'importe quel mot w de taille n, la profondeur de toute branche de $T(M, w)$ est inferieure ou égale à $f(n)$.

8.2 classes de complexité

On dénote la famille des langages acceptés par des machines de complexité d'espace $f(n)$ par DESPACE$(f(n))$ et par NESPACE$(f(n))$ si les machines sont non-déterministes. De même, on dénote la famille des langages acceptés par des machines de complexité de temps $f(n)$ par DTEMPS$(f(n))$ et par NTEMPS$(f(n))$ si les machines sont non-déterministes.

compression, accéleration, réduction

Quand on dit qu'un langage L a une complexité en espace $f(n)$ ce que l'on veut vraiment dire est qu'il existe une machine M qui accepte L en utilisant au maximum un nombre de cases lié à la taille de la donnée par une fonction dont l'*ordre de grandeur* est celui de $f(n)$; les facteurs constant sont ignorés. Cette approche est justifiée par le résultat suivant.

Théorème 8.1 *(***compression de l'espace***)*
i) Si L est accepté par une machine déterministe bornée en espace par $f(n)$ avec k rubans de travail, alors pour tout réel $c > 0$, L est accepté par une machine déterministe bornée en espace par $cf(n)$.
ii) Le même résultat est valable dans le cas non déterministe.
iii)

$$\forall c > 0 \;\; [\textsc{Despace}(f(n)) = \textsc{Despace}(cf(n))] .$$

$$\forall c > 0 \;\; [\textsc{Nespace}(f(n)) = \textsc{Nespace}(cf(n))] .$$

Preuve Commençons par prouver (i). Si $c \geq 1$, le résultat est immédiat, car alors $f(n) \leq cf(n)$. Le seul cas interessant est donc celui où $0 < c < 1$: par exemple, pour $c = \frac{1}{2}$, on veut comprimer l'espace de la moitié...

Soit donc M_1 une machine déterministe bornée en espace par $f(n)$. On montre d'abord que $\forall$ entier positif r, il existe une machine déterministe M_2 qui simule M_1 en codant dans chaque case d'un ruban de travail r symboles consecutifs du ruban de travail correspondant de M_1.

Par simplicité de notation, on va supposer que M_1 a un seul ruban de travail, donc que $k = 1$, mais la construction se généralise de façon évidente au cas de plusieurs rubans. Soit Σ_1 l'alphabet de M_1; l'alphabet Σ_2 de M_2 contient tous les mots sur Σ_1 de longeur r. Les états de M_2 sont des couples $< q, i >$, où q est un état de M_1 et $1 \leq i \leq r$; la composante i sert à mémoriser le symbole lu par M_1 parmi les r symboles consecutifs correspondant à la case lue par M_2. Par example, si q_1 est l'état initial de M_1, la machine M_2 démarre dans l'état $< q_1, 1 >$. Soit :

$$\delta_1(q, \sigma) = (q', \gamma, \rhd)$$

une transition de M_1. Plusieurs transitions de M_2 lui sont associées :
Toutes les transitions de la forme :
$\delta_2(< q, 1 >, m^1) = (< q', 2 >, w^1, \bigtriangledown)$
où m^1 est n'importe quel mot de longeur r sur Σ_1 dont le premier symbole est σ et w^1 est le mot obtenu à partir de m^1 en remplaçant σ par γ.
$\vdots$
Toutes les transitions de la forme :
$\delta_2(< q, r-1 >, m^{r-1}) = (< q', r >, w^{r-1}, \bigtriangledown)$
où m^{r-1} est n'importe quel mot de longeur r sur Σ_1 dont l'avant-dernier symbole est σ et w^{r-1} est le mot obtenu à partir de m^{r-1} en remplaçant σ par γ.
Toutes les transitions de la forme :
$\delta_2(< q, r >, m^r) = (< q', 1 >, w^r, \rhd)$
oú m^r est n'importe quel mot de longeur r sur Σ_1 dont le dernier symbole est σ et w^r est le mot obtenu à partir de m^r en remplaçant σ par γ.

Les transitions de M_2 pour les cas des mouvements à gauche et sur place sont construites de façon semblable. Il est clair que M_2 simule M_1 en occupant au maximum $\lceil \frac{f(n)}{r} \rceil$ cases. Or, si l'on choisit r tel que $r \cdot c \geq 2$ on obtient le résultat volu.

La preuve de la partie (ii) du théorème est identique à celle de (i), car la même construction est valable dans le cas non déterministe.

Enfin, les égalités entres classes de complexité de la partie (iii) suivent de façon évidente de (i) et (ii).
$\blacksquare$

De la même façon, quand on dit qu'un langage a une complexité en temps $f(n)$, ce que l'on veut dire est qu'il est reconnu par une machine dont le nombre de transitions est borné par une fonction de l'*ordre de grandeur* de $f(n)$. En fait, le résultat suivant nous dit que, sous certaines conditions, on peut toujours accélérer une machine d'un facteur constant.

Théorème 8.2 *(***accélération linéaire***).*
i) Si L est accepté par une machine bornée en temps par $f(n)$ avec $k > 1$ rubans de travail, alors pour tout réel $c > 0$, L est accepté par une machine bornée en temps par $cf(n)$, à condition que

$$\lim_{n\to\infty} \frac{f(n)}{n} = \infty.$$

ii) Le même résultat est valable dans le cas non déterministe.
iii) On a donc :

$$\lim_{n\to\infty} \frac{f(n)}{n} = \infty \Rightarrow \forall c > 0 \ [\textsc{Dtemps}(f(n)) = \textsc{Dtemps}(cf(n))].$$

$$\lim_{n\to\infty} \frac{f(n)}{n} = \infty \Rightarrow \forall c > 0 \ [\textsc{Ntemps}(f(n)) = \textsc{Ntemps}(cf(n))].$$

Preuve. On commence par démontrer un résultat intermédiaire. Soit M_1 une machine déterministe bornée en temps par $f(n)$ avec $k > 1$ rubans de travail et soit r un entier positif quelconque. On peut alors construire une machine déterministe M_2 qui simule M_1 (et accepte donc le même langage) bornée en temps par $n + \lceil \frac{n}{r} \rceil + 8\lceil \frac{f(n)}{r} \rceil$. La preuve de ce résultat est à faire comme exercice en TD.

Maintenant, prenons r tel que $c \cdot r \geq 16$ et appelons A l'expression $n + \lceil \frac{n}{r} \rceil + 8\lceil \frac{f(n)}{r} \rceil$. Puisque $\lceil x \rceil < x + 1$ on obtient :

$$A \leq n + \frac{n}{r} + 8\frac{f(n)}{r} + 9 \tag{1}$$

Puisque on a supposé que $\liminf_{n\to\infty} \frac{f(n)}{n} = \infty$ on a :
$\forall$ constante $d, \exists n_d$ tel que $\forall n \geq n_d, \frac{f(n)}{n} \geq d$, en autre mots, $n \leq \frac{f(n)}{d}$.
Donc, $\forall n$ tel que $n \geq 9$ (d'oú $n + 9 \leq 2n$) et $n \geq n_d$, on a d'abord :

$$A \leq 2n + \frac{n}{r} + 8\frac{f(n)}{r} \tag{2}$$

puis :

$$A \leq 2\frac{f(n)}{d} + \frac{f(n)}{dr} + 8\frac{f(n)}{r} \tag{3}$$

d'oú :

$$A \leq f(n)(\frac{2}{d} + \frac{1}{dr} + \frac{8}{r}) \tag{4}$$

Soit $d = \frac{r}{4} + \frac{1}{8}$. Puisque on a choisi r tel que $c \cdot r \geq 16$, si l'on remplace r par $\frac{16}{c}$, on obtient que $\exists n_d$ tel que $\forall n \geq max(9, n_d)$ on a :

$$A \leq cf(n) \tag{5}$$

En ce qui concerne les mots de longueur n inferieure à $max(9, n_d)$, il n'y en a qu'un nombre fini. Donc on peut modifier M_2 de façon à les gérer en utilisant exclusivement son controle fini (des états *ad hoc*); on n'aura besoin que de $n+1$ transitions pour lire la donnée et réperer le blanc à la fin. Donc M_2 est en tous cas bornée en temps par $\max(n+1.cf(n))$ (ce qui revient à dire par $cf(n)$ vu la covention adoptée).

La construction pour le cas non-déterministe (ii) est la même. Enfin, les égalités de (iii) suivent de (i) et (ii).
■

Il est aussi possible de réduire le nombre des rubans d'une machine tout en gardant le contrôle sur le temps et l'espace utilisés : ceci nous est dit par le résultat suivant, qui est valable pour le cas déterministe aussi bien que pour le cas non-déterministe.

Théorème 8.3 *(***réduction du nombre de rubans***).*
i) Si L est accepté par une machine bornée en espace par $f(n)$ avec k rubans de travail, alors L est accepté par une machine bornée en espace par $f(n)$ n'utilisant qu'un seul ruban de travail.
ii) Si L est accepté par une machine bornée en temps par $f(n)$ avec k rubans de travail, alors L est accepté par une machine bornée en temps par $f^2(n)$ n'utilisant qu'un seul ruban de travail.

Idée de preuve: Il suffit de considérer la technique décrite à la section 3.3 visant à simuler une machine avec plusieurs rubans et têtes de lectures par une machine à une seule. Cette technique nous donne directement ces deux résultats.

8.3 Hiérarchies en espace et en temps

Les classes de complexité en espace et en temps, déterministe ou non déterministe, sont structurées en hiérarchies. Par exemple, il est évident que si $f_1, f_2, \cdots f_n$ sont des fonctions d'ordre de grandeur croissant, on a les inclusions :

$$\textsc{Despace}(f_1(n)) \subseteq \textsc{Despace}(f_2(n)) \cdots \subseteq \textsc{Despace}(f_n(n))$$

De même pour le temps déterministe et pour les hiérarchies non déterministes. On peut se poser la question s'il existe une classe de complexité en temps déterministe qui contient *tous* les langages décidables. Le résultat suivant nous dit que ceci est faux et qu'il existe une hiérarchie *infinie* de classes de complexité en temps déterministe; de même, il existe une hiérarchie *infinie* de classes de complexités en espace déterministe. On a des résultats analogues pour la complexité en temps non déterministe et pour la complexité en espace non déterministe.

Théorème 8.4 *Étant donnée une fonction récursive $f(n)$, il existe toujours un langage récursif L tel que $L \notin$ DTEMPS$(f(n))$. De même, il existe toujours un langage récursif L' tel que $L \notin$ DESPACE$(f(n))$*

Preuve Dans le cas du temps aussi bien que dans le cas de l'èspace, il s'agit de faire une argumentation par diagonalisation; nous détaillons ici exclusivement le cas du temps.

D'abord, rappellons que l'on peut énumerer de façon effective tous les mots sur l'alphabet $\{0,1\}$; on peut donc parler de x_i, le i-ème mot dans cette énumeration. Puisque on peut coder par des mots sur $\{0,1\}$ toutes les machines de Turing déterministes (avec n'importe quel nombre de rubans) et énumerer de façon effective tous ces codages, on peut aussi parler de M_i, la i-ème machine selon cette énumeration.

Puisque f est récursive (totale), il existe une machine M_f qui la calcule (en s'arrêtant toujours). Maintenant, soit le langage :
$L = \{x_i | x_i \in \{0,1\}^*$ et M_i n'accepte pas x_i en $f(|x_i|)$ transitions $\}$.
Ce langage est récursif : voici un algorithme qui peut être suivi par une machine de Turing pour décider si un mot $w \in L$. Etant donnée une donnée w de longeur n, d'abord on simule M_f pour calculer $f(n)$. Puis on détermine l'entier i tel que w est x_i. Ensuite, on repère la machine M_i dans l'énumeration des machines et on simule M_i sur w pour $f(n)$ transitions : on rejette w si M_i s'arrête en acceptant w, sinon on accepte.

Montrons par l'absurde que $L \notin$ DTEMPS$(f(n))$. Supposons le contraire : il existe alors une machine qui accepte L et qui est bornée en temps par $f(n)$; soit j l'index de cette machine. On va montrer qu'alors $x_j \in L$ si et seulementi si $x_j \notin L$, une contradiction. Si $x_j \in L$, alors, puisque M_j accepte L en $f(|x_j|)$ transitions, il en suit que $x_j \notin L$, par définition de L. Réciproquement, si $x_j \notin L$, M_j n'accepte pas x_j, donc, par définition de L, il en suit que $x_j \in L$.

On a donc montré l'existence d'un langage récursif L qui n'est pas dans $\textsc{Dtemps}(f(n))$.
■

Le résultat precedant nous assure qu'il n'y a pas de classe maximale de complexité déterministe en temps, et de même pour l'espace. Maintenant, on peut se poser la question de combien faut-il incrementer une fonction f pour obtenir une fonction f' telle que $\textsc{Despace}(f(n))$ soit proprement incluse dans $\textsc{Despace}(f'(n))$, c'est à dire telle que l'on ait

$$\exists L(L \in \textsc{Despace}(f'(n)) \wedge L \notin \textsc{Despace}(f(n))).$$

Or, si f respecte une certaine condition, un petit incrément de f est suffisant pour obtenir une nouvelle classe. Nous allons d'abord préciser cette condition.

Définition 8.5 *Une fonction f est dite complètement constructible en espace s'il existe une machine de Turing M^f telle que $\forall n, \forall$ donnée w de taille n, M^f utilise (marque) exactement $f(n)$ cases.*

L'interêt de la notion de fonction *complètement constructible en espace* est qu'elle permet de fixer une borne sur l'espace qui doit être utilisé par une machine. Autrement dit, si on sait qu'une fonction f est complètement constructible en espace par une machine M^f, et on veut qu'une machine M soit bornée en espace par $f(n)$, on peut "initialiser" M en utilisant M^f pour "allouer" les $f(n)$ cases mémoire que M n'aura pas le droit de dépasser dans le calcul ultérieur. L'ensemble des fonctions complètement constructibles en espace est très riche et comprend, par exemple, $\log n, 2^n, n^k, n!$...

Remarque

La définition ci-dessus ne dit pas explicitement que la machine M_f qui construit f en espace s'arrête tout le temps. Toutefois, on peut supposer sans perte de généralité que ceci est le cas, car, étant donnée une machine M_f qui construit f en espace et qui peut ne pas s'arrêter, on peut en construire une autre, disons T_f, qui fait la même chose mais qui s'arrête tojours.

Théorème 8.6 *Si f_2 est totalement constructible par rapport à l'espace, si $f_1(n), f_2(n) \geq \log_2 n$ et si*

$$\liminf_{n\to\infty} \frac{f_1(n)}{f_2(n)} = 0$$

alors $\textsc{Despace}(f_2(n)) \not\subseteq \textsc{Despace}(f_1(n))$.

On a un résultat analogue pour le temps déterministe. Avant de l'énoncer, nous avons besoin d'une nouvelle définition.

Définition 8.7 *Une fonction f est dite complètement constructible en temps s'il existe une machine de Turing M^f telle que $\forall n, \forall$ donnée w de taille n, M^f fait exactement $f(n)$ transitions.*

L'interêt de la notion de fonction *complètement constructible en temps* est qu'elle permet de fixer une borne sur le temps utilisé par une machine. En faisant marcher "en parallèle" une machine M^f qui construit f en temps et une machine M, on peut utiliser la première comme un "réveil" pour la deuxième, de façon telle que M s'arrête quand "le réveil sonne"... Ceci permettra d'obtenir que M ne dépasse pas $f(n)$ transitions.

Le résultat analogue au théorème 8.6 pour le temps est aussi énoncé sans preuve.

Théorème 8.8 *Si f_2 est complètement constructible par rapport au temps et si*

$$\liminf_{n\to\infty} \frac{f_1(n)\log f_1(n)}{f_2(n)} = 0$$

alors

$$\textsc{Dtemps}(f_2(n)) \not\subseteq \textsc{Dtemps}(f_1(n)).$$

8.4 Relations entre les mesures de complexité

Théorème 8.9
1. *Si $L \in \textsc{Despace}(f(n))$ alors $L \in \textsc{Nespace}(f(n))$ et si $L \in \textsc{Dtemps}(f(n))$ alors $L \in \textsc{Ntemps}(f(n))$.*
2. *Si $L \in \textsc{Ntemps}(f(n))$ et f est complètement constructible en temps alors $L \in \textsc{Despace}(f(n)$.*

3. *Si* $L \in \textsc{Nespace}(f(n))$, *$f$ est complètement constructible en espace et si* $f(n) \geq \log_2 n$ *alors il existe un* $c_L > 0$ *tel que* $L \in \textsc{Dtemps}(c_L^{f(n)})$.

4. *Si* $L \in \textsc{Ntemps}(f(n))$ *et si f est complètement constuctible en temps, alors il existe un* $c_L > 0$ *tel que* $L \in \textsc{Dtemps}(c_L^{f(n)})$.

La preuve de ces résultats est laisseé en exercice (TD). Le prochain résultat est beaucoup plus délicat; sa preuve sert aussi à clarifier l'idèe intuitive qui est derriere la notion de constructibilité en espace d'une fonction f.

Théorème 8.10 (Savitch)
Si $L \in \textsc{Nespace}(f(n))$ *alors* $L \in \textsc{Despace}(f^2(n))$ *à condition que* $f(n)$ *soit complètement constructible en espace et que* $f(n) \geq \log_2 n$.

Preuve. Soit M_1 la machine non déterministe bornée en espace par une fonction $f(n) \geq \log_2 n$. Il existe alors une constante c telle que le nombre de configurations distinctes de M_1 est borné par $c^{f(n)}$ (exercice en TD). Donc M_1 accepte un mot w de longeur n si et seulement s'il existe une chaîne de configurations :

$$C_{i_1} \vdash C_{i_2} \cdots, C_{i_{k-1}} \vdash C_{i_k}$$

telle que

- C_{i_1} est la configuration initiale correspondant à l'entrée w,
- C_{i_k} est une configuration acceptante (elle contient un état d'acceptation),
- $k \leq c^{f(n)}$.

On va donc construire une machine déterministe M_2 qui simule M_1 en testant l'existence d'une telle chaîne pour l'entrée w considerée. Le problème va être comment faire ceci avec "peu d'espace".

L' inégalité $f(n) \geq \log_2 n$ permet de représenter les configurations par d'une part la position de la tête de lecture sur le mot d'entrée, et d'autre part le contenu du ruban de travail, le tout sans dépasser $f(n)$.

On va d'abord présenter une argumentation informelle, sur un exemple simple, pour donner une intuition sur le fonctionnement de M_2 et sur la raison pour laquelle l'espace mémoire utilisé est de l'ordre de grandeur de

$f^2(n)$. Ensuite, nous donnerons une description precise de l'algorithme suivi par M_2 et du calcul qui montre que la borne en espace est effectivement celle voulue.

Soit par exemple M_1 telle que $c = 3$ et $f(x) = 2x$ et soit 1 la longeur de w. On a donc 9 configurations possibles et une chaîne de configurations ne comportant pas de répetitions contient au maximum 8 transitions. Il faut donc examiner l'une apres l'autre toutes les configuration acceptantes et vérifier s'il en existe au moins une qui est le point d'arrivée d'une telle chaîne. Notons par $C' \stackrel{\leq j}{\longmapsto} C''$ le fait que M_1 permet de passer de C' à C'' avec j transitions au maximum. Soit C_a une configuration acceptante. Pour tester si $C_{i_1} \stackrel{\leq 8}{\longmapsto} C_a$, la machine M_2 teste s'il existe une configuration "centrale", c'est à dire une C_{i_5} telle que $C_{i_1} \stackrel{\leq 4}{\longmapsto} C_{i_5}$ **et** $C_{i_5} \stackrel{\leq 4}{\longmapsto} C_a$. Pour cela, il faut examiner l'une aprés l'autre les 9 configurations et verifier s'il en a au moins une qui convient. Or, pour tester si une configuration particulière candidate à être C_{i_5} est "bonne", on teste s'il existe une C_{i_3} et une C_{i_7} telles que :

- $C_{i_1} \stackrel{\leq 2}{\longmapsto} C_{i_3}$ **et** $C_{i_3} \stackrel{\leq 2}{\longmapsto} C_{i_5}$
- $C_{i_5} \stackrel{\leq 2}{\longmapsto} C_{i_7}$ **et** $C_{i_7} \stackrel{\leq 2}{\longmapsto} C_a$

etc. On procéde donc par *dichotomie*, jusqu'à trouver, si elle existe, la chaîne voulue.

Ceci revient à explorer, pour toute C_a, un arbre AND-OR; pour chercher la bonne chaîne on fait un parcours en profondeur avec *backtracking* de cet arbre. Ce qu'il est important d'observer est qu'on n'aura jamais besoin de garder l'arbre entier en memoire : il suffit de garder, à chaque coup, une seule branche. Ceci revient à dire que le ruban de travail va contenir au maximum autant de configuration qu'il en a dans une branche. En particulier, quand l'on veut savoir si $C' \stackrel{\leq 2j}{\longmapsto} C''$ et on a déjà verifié que $C' \stackrel{\leq j}{\longmapsto} C'''$, on peut effacer le contenu du ruban et *réutiliser cet espace* pour verifier si $C''' \stackrel{\leq j}{\longmapsto} C''$.

Or, combien de configurations apparaissent dans une branche? Dans notre exemple, elles sont 3. Bien évidemment, pour faire le test aux feuilles on peut avoir aussi besoin de la configuration initiale ou de celle acceptante; cela fait 4 configurations à mémoriser pour étudier une branche. On observera que, pour les valeurs de c et $f(n)$ de notre exemple, 4 est le résultat de la multiplication de $f(n)$ par $\lceil \log_2 c \rceil$. Or, cette relation est vraie en général. Elle nous dit que le nombre de configurations qu'il faut mémoriser sur le ruban de travail n'excede jamais l'ordre de grandeur de $f(n)$. La taille de chaque configuration étant de l'ordre de grandeur de $f(n)$ (car M_1 n'utilise jamais plus que $f(n)$ cases), on voit bien que la borne en espace de M_2 est de l'ordre de grandeur de $f^2(n)$.

Donnons maintenant l'algorithme precis suivi par M_2; l'essentiel du travail est fait par la procédure TEST.

Entrée : w.
begin
n:=$|w|$; $m := \lceil \log_2 c \rceil$;
C_{i_1}:= Configuration initiale de M_1 pour l'entrée w;
pour toute C_a acceptante de longeur maximale $f(n)$ **faire**
  si TEST(C_{i_1}, C_a, $mf(n)$) **alors accepte**;
end;

procedure TEST(C', C'', j);
  si $j = 0$ et ($C' = C''$ ou $C' \vdash C''$ **alors return VRAI**;
  si $j \geq= 1$ **alors**
    pour toute C''' de longeur maximale $f(n)$ **faire**
      si TEST(C', C''', j-1) et TEST(C''', C'', j-1) **alors**
        return VRAI;
  return FAUX
end TEST

La procédure TEST a trois arguments : deux configurations et un entier j; elle vérifie si l'on peut passer de la premiere configuration à la deuxieme avec 2^j transitions au maximum. Pour ceci, elle cherche une configuration que l'on puisse placer "au milieu" et teste recursivement si elle " marche". Ceci revient à verifier l'existence de **deux** chaînes de transitions de longeur maximale $\frac{2^j}{2}$. Ma, comme on a déjà observé, *le test pour l'existence d'une des deux chaînes, disons celle de droite, peut réutiliser l' espace déjà utilisé pour l'autre (celle de gauche).* Ceci permet d'utiliser le ruban de travail comme un pile de "blocs" où on memorise les appels récursifs de TEST et n'avoir jamais plus que $mf(n)$ appels à memoriser dans cette pile (chaque appel décremente j de 1). Le nombre de blocs de la pile est donc de l'ordre de $f(n)$.

Dans chaque bloc il faut mémoriser :

- les valeur courantes de C', C'' et C'''. La taille en espace de chacune d'elles est de l'ordre de grandeur de $f(n)$,
- la valeur courante de j; puisque j va de $mf(n)$ à 1, où m est une constante, ceci se fait en notation binaire avec un espace dont l'ordre de grandeur est $\log_2 f(n)$.

Donc l'espace nécessaire pour gérer la pile est bien $f^2(n)$.

Observer que l'hypothèse $f(n) \geq \log_2 n$ sert à assurer la possibilité de répresenter la position de la tête de lecture sur le ruban d'entrée sans dépasser la borne d'espace volue.

Enfin, observer que l'hypothèse que $f(n)$ est complètement constructible en espace sert à *allouer* l'espace mémoire pour les $f(n)$ blocs de la pile - chacun de taille $f(n)$- correspondant aux appels recursifs de TEST; elle sert aussi à connâitre en avance la valeur de $mf(n)$ qui permet de faire demarrer le programme. Il ne s'agit pas d'une hypothèse vraiment contraignante, car, comme on l'a dit, pratiquement toutes le fonctions que l'on manipule couramment sont constructibles en espace!

Il interessant de remarquer que, en l'état actuel des connaissances, on ne sait pas si la borne $f^2(n)$ donnée par ce résultat est optimale ou pas.

8.5 Autres résultats sur les mesures de complexité

Il existe des résultats de complexité qui sont assez "surprenants", dans le sens où ils semblent aller contre l'intuition. Ici nous en donnons deux, sans les prouver.

Les théorèmes 8.6 et 8.8 que l'on a déjà vu nous disent que les hiérarchies en espace et et en temps sont très denses, car étant donnée une classe de complexité $f_1(n)$, il suffit qu'une autre fonction $f_2(n)$ soit "juste un peu" plus rapide que f_1 pur avoir une nouvelle classe de complexité. Toutefois, observez que ces résultats ne sont valides que sous l'hypothése que la fonction f_2 soit complétement constructible (respectivement, en espace et en temps). Si on supprime cette hypothése, on obtient qu'on peut avoir une "brèche" arbitrairement grande entre deux classes : on peut incrementer autant que l'on veut f_1 sans sortir de la classe de complexité correspondant à f_1. Ceci nous est dit par le théorème suivant.

Théorème 8.11 (Brèche de Borodin) *i) Étant donné n'importe quelle fonction récursive $g(n) \geq n$, il existe une fonction récursive $f(n)$ telle que*

$$\text{DESPACE}\,(f(n)) = \text{DESPACE}\,(g(f(n)))$$

ii) même chose pour l'espace non déterministe, pour le temps déterministe et pour le temps non déterministe.

Le théorème 8.2 déjà vu nous dit que, étant donné *n'importe quel* langage L, on peut toujours accélerer d'une *constante* le temps d'une machine acceptant L. Or, le résultat suivant nous dit qu'il *existe des langages* tels que on peut accélerer les machines les acceptant *autant que l'on veut*. Par exemple, il existe un langage L' tel que, si M_1 accepte L' avec une complexité en temps $f(n)$, on peut modifier M_1 de façon à avoir une complexité $\sqrt{f_1(n)}$, puis la modifier à nouveau pour obtenir une complexité $\sqrt{\sqrt{f_1(n)}}$, etc. Ceci nous est assuré par le théorème suivant, où l'expression *presque partout* signifie "$\forall$ valeur de n sauf que pour un nombre fini".

Théorème 8.12 (Accéleration de Blum) *Étant donné n'importe quelle fonction récursive $g(n)$, il existe un langage récursif L tel que pour toute machine M_i acceptant L avec en espace $f_i(n)$ (ou en temps $f_i(n)$) il existe une machine M_j acceptant L en temps $f_j(n)$ où*

$$g(f_j(n)) \leq f_i(n) \textit{ presque partout}$$

Nous passons maintenant à une branche de la théorie de la complexité qui est centrée sur la notion de "faisabilité" de problèmes.

9 “Traitabilité” de problèmes de décision et $\mathcal{NP}$-complétude

Nous abordons maintenant l'étude des classes de complexité jugées comme les plus importantes d'un point de vu théorique: $\mathcal{P}$ et $\mathcal{NP}$.

9.1 les classes $\mathcal{P}$ et $\mathcal{NP}$

Définition 9.1

$$\mathcal{P} = \bigcup_{i \in N} \text{DTEMPS}(n^i)$$

$$\mathcal{NP} = \bigcup_{i \in N} \text{NTEMPS}(n^i)$$

Remarquez que ces deux classes sont stables par rapport aux changement de modèle de machine de Turing (en excluant le non-déterminisme, bien sûr). Par exemple, par le théorème 8.3, le nombre de rubans de la machine utilisée ne peut faire qu'un changement quadratique dans le temps nécessaire pour accepter un langage.

Donnons deux exemples typiques de problèmes très célèbres (intervenant dans de nombreuses applications) et qui sont dans les classes $\mathcal{P}$ et $\mathcal{NP}$.

Programmation Linéaire . C'est un problème typique de la classe $\mathcal{P}$.

donnée: m vecteurs d'entiers $V_i = (v_i[1], \ldots, v_i[n])$, $1 \leq i \leq m$, deux vecteurs d'entiers $D = (d[1], \ldots, d[m])$, $C = (c[1], \ldots, c[n])$ et un entier B

question: existe-t-il un vecteur $X = (x[1], \ldots, x[n])$ de nombres rationnels tels que $C \cdot X \geq B$ et $V_i \cdot X \leq d[i]$ pour tout i?

Problème du voyageur de commerce . C'est un problème typique de la classe $\mathcal{NP}$.

donnée: une liste $v_1, \ldots, v_n$ de *villes*, une *distance* entière $d(v_i, v_j)$ pour toutes les paires de villes distinctes et une borne B.

question: existe-t-il un chemin passant par toutes les villes, revenant au départ et dont la longeur est bornée par B ?

Les deux problèmes ci-dessus sont de nature fondamentalement différente. Pour bien le comprendre, nous allons montrer un problème qui possède les deux facettes, selon la question posée. Considérons le problème S de savoir si un ensemble de clauses $\{C_1 \cdots C_n\}$ de la logique propositionnelle est vrai pour une interpretation I *donnée*. Comme on sait, il s'agit d'un problème décidable, donc l'ensemble

$$E = \{< \{C_1 \cdots C_n\}, I > \mid \forall i, 1 \leq i \leq n,\ I(C_i) = \ Vrai\}$$

est récursif. Il est clair que, une fois choisi un codage approprié pour les données de la forme $< \{C_1 \cdots C_n\}, I >$, il existe une machine de Turing déterministe qui décide si une telle donnée appartient à E avec une complexité de temps $f(N)$, où N est le nombre total d'occurrences de littéraux dans $C_1, \ldots, C_n$, et f est une fonction polynomiale (en fait linéaire) de N. Il existe donc un *algorithme déterministe polynomial* qui décide le problème S, ce que l'on exprime en disant que $S \in \mathcal{P}$.

Maintenant, considérons le problème de savoir, étant donné un ensemble de clauses $\{C_1 \cdots C_n\}$ propositionnelles, s'il*existe* au moins une interprétation I pour laquelle toutes les clauses sont vraies. Ce problème est connu sous le nom de $\mathcal{SAT}$, ou problème de la satisfaisabilité (propositionnelle). Il existe un algorithme déterministe évident pour décider ce problème : on teste une par une toutes les interprétations possibles. Malheuresement, il en existe 2^k, où k est le nombre des variables propositionnelles de l'ensemble de clauses. Dans le pire des cas, k pourrait être égal à N (nombre total d'occurrences de littéraux).[2] Donc cet algorithme n'est pas polynomial. Mais il existe un algorithme *non déterministe* qui "donne une réponse" au problème $\mathcal{SAT}$ en temps polynomial :

- choisir une interprétation I
- vérifier que I rend vraies toutes les clauses (ce qui se fait en temps polynomial).

[2] Remarquer que, à l'inverse, N peut être de l'ordre de 2^k puisque chaque clause peut être vu comme un sous-ensemble arbitraire de l'ensemble des littéraux.

En autres mots, une fois codé de façon appropriée, le langage

$$L_{sat} = \{\{C_1 \cdots C_n\} | \; \exists I \forall i, 1 \leq i \leq n, \; I(C_i) = \; Vrai\}$$

est accepté par une machine de Turing non-déterministe avec une complexité de temps polynômial. Ceci s'exprime en disant que $SAT \in \mathcal{NP}$.

Intuitivement, la classe $\mathcal{P}$ est la classe des problèmes que l'on peut décider "efficacement" (par exemple le problème S). La classe $\mathcal{NP}$ est la classe des problèmes dont on sait seulement vérifier efficacement si une "solution candidate marche" (par exemple, le problème SAT, où une "solution candidate" est une interprétation I particulière, que l'on a choisi de façon non-déterministe).

Une question de toute première importance en informatique théorique est :

$$\mathcal{P} \stackrel{?}{=} \mathcal{NP}.$$

Il est clair que $\mathcal{P} \subseteq \mathcal{NP}$, mais que dire de l'inclusion réciproque ? L'intuition semble suggérer qu'elle est fausse, car il est plus difficile de trouver la solution correcte d'une question que de vérifier si une solution donnée est juste ou pas. Toutefois, cette conjecture reste un problème ouvert... Une preuve de $\mathcal{P} \neq \mathcal{NP}$ aurait des implications pratiques importantes car des nombreux problèmes naturels ne peuvent être résolus efficacement (polynômialment) que si $\mathcal{P} = \mathcal{NP}$; en un certain sens, ces problèmes ne sont pas "faisables" si $\mathcal{P} \neq \mathcal{NP}$.

9.2 la classe $\mathcal{P}$ESPACE

On peut définir de façon semblable de telles classes par rapport à la complexité en espace:

Définition 9.2

$$\mathcal{P}\textsc{espace} = \bigcup_{i \in N} \textsc{Despace}(n^i)$$

$$\mathcal{NP}\textsc{espace} = \bigcup_{i \in N} \textsc{Nespace}(n^i)$$

Mais cette fois, pas de grande question ouverte car par le théorème de Savitch (8.10), il découle que

$$\mathcal{P}\textsc{espace} = \mathcal{NP}\textsc{espace}.$$

9.3 Relations sur ces classes

On a les relations suivantes avec certitude sur ces classes:

$$\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{P}\textsc{espace}$$

mais on ne sais pas si ces inclusions sont strictes ou non. L' inclusion $\mathcal{NP} \subseteq \mathcal{P}\textsc{espace}$ est une consequence du théoreme 8.9 et du fait que $\mathcal{P}\textsc{espace} = \mathcal{NP}\textsc{espace}$. En fait, on ne sait même pas si l'inclusion $\mathcal{P} \subseteq \mathcal{P}\textsc{espace}$ est stricte.

9.4 Réductions polynômiales

On introduit maintenant la notion de réduction polynômiale entre des langages récursifs. L'intuition qui est derriere cette notion est que si un problème de décision π_1 se réduit polynômialment à un probléme π_2 et l'on a un algoritme polynômial qui décide le deuxième, alors on a aussi un algorithme polynômial qui décide le premier, car on peut toujours rammener π_1 à π_2 (sans sortir du temps polynômial).

Définition 9.3 *On dit que $L_1 \subseteq \Sigma_1^*$ se réduit polynômialement à $L_2 \subseteq \Sigma_2^*$ (noté $L_1 \leq_{\mathcal{P}} L_2$) s'il existe une fonction $f : \Sigma_1^* \to \Sigma_2^*$ telle que*

1. *f est calculable en temps polynômial par rapport à la taille de sa donnée*
2. *pour tout $x \in \Sigma_1^*$ on a que*

$$x \in L_1 \Leftrightarrow f(x) \in L_2$$

Théorème 9.4 *Si $L_2 \in \mathcal{P}$ et $L_1 \leq_{\mathcal{P}} L_2$ alors $L_1 \in \mathcal{P}$.*

Soit p_2 un polynôme bornant le temps nécessaire pour reconnaître L_2, soit f une fonction servant à réduire L_1 à L_2 et soit p un polynôme bornant son temps d'exécution. On pourra décider pour chaque mot x s'il appartient ou non à L_1 en temps $p(|x|) + p_2(p(|x|))$ (on prend un temps $p(|x|)$ pour calculer $f(x)$ et un temps $p_2(|f(x)|)$ pour décider si $f(x) \in L_2$, où $|f(x)| \leq p(|x|)$).

Utilisation d'Oracles

Remarquons que si $L_1 \leq_P L_2$, alors L_1 est décidé en temps polynômial par une machine de Turing M^{L_2}, avec un oracle décidant l'appartenance à L_2. Mais la réciproque n'est pas correcte car:

1. en général, M^{L_2} pourrait interroger plusieurs fois l'oracle
2. M^{L_2} pourrait effectuer des transitions après avoir consulté l'oracle.

En fait, il y a bien équivalence entre ces deux définitions, pourvu que M_{L_2} n'ait le droit de consulter qu'une fois l'oracle, et uniquement lors de la dernière transition.

Autres réductions

Nous avons vu ci-dessus les réductions polynômiales, mais la définition 9.3 s'étend à n'importe quelle classe de complexité. Par exemple, on peut considérer les réductions qui se font en espace logarithmique (et, partant, la relation $\leq_{\text{LOGSPACE}}$).

9.5 $\mathcal{NP}$-complétude

On a déjà dit que nombreux problèmes ne sont pas "faisables" si $\mathcal{P} \neq \mathcal{NP}$; il s'agit des problèmes que l'on appelle $\mathcal{NP}$-complets.

Définition 9.5 *Un langage L est dit* $\mathcal{NP}$-complet *si ces deux conditions sont satisfaites :*

1. $L \in \mathcal{NP}$
2. *pour tout langage $L' \in \mathcal{NP}$, $L' \leq_{\mathcal{P}} L$ (L est* $\mathcal{NP}$-difficile*)*

Beaucoup de problèmes pratiques, par exemple un grand nombre de problèmes de graphes, sont connus comme étant $\mathcal{NP}$-complets.

Si un problème π (un langage L_π) est $\mathcal{NP}$-complet et $\mathcal{P} \neq \mathcal{NP}$, alors $\pi \in \mathcal{NP}$ mais $\pi \notin \mathcal{P}$, car autrement, puisque tout élement de $\mathcal{NP}$ se réduit à π, on aurait que $\mathcal{P} = \mathcal{NP}$.

Observez qu'un problème peut être dans $\mathcal{NP}$ sans être *$\mathcal{NP}$-difficile* ou bien être *$\mathcal{NP}$-difficile* sans être dans $\mathcal{NP}$; dans aucun de ces deux cas le problème n'est NP-complet.

Les problèmes $\mathcal{NP}$-complets sont historiquement considérés comme étant des problèmes qui n'ont pas de solution efficace. La raison en est que, si $\mathcal{P} \neq \mathcal{NP}$, ces problèmes ne peuvent être résolus en temps polynomial (dans le pire des cas). Cependant, il existe d'autres notions de complexité qui modélisent d'autres aspects de l'efficacité d'un algorithme. En particulier, la *complexité en moyenne* qui ne considère pas seulement le pire des cas, mais l'ensemble des données et, le cas échéant, une probabilité d'apparition pour chacune de ces données. Bien entendu, un problème peut être $\mathcal{NP}$-complet tout en étant polynomial du point de vue de la complexité en moyenne.

La plus grande difficulté avec la $\mathcal{NP}$-complétude fut de démontrer l'existence d'un premier langage $\mathcal{NP}$-complet. Par contre, une fois ce premier langage obtenu, il est beaucoup plus simple de démontrer la même propriété en utilisant le lemme suivant:

Lemme 9.6 *Un langage L est* $\mathcal{NP}$-complet *si*

1. $L \in \mathcal{NP}$
2. *il existe un langage* $\mathcal{NP}$-complet L' *tel que* $L' \leq_{\mathcal{P}} L$

L'exemple classique de problème *$\mathcal{NP}$-complet* est le problème SAT déjà discuté informellement. C'est le premier problème dont on a démontré la $\mathcal{NP}$-complétude.

Dans la suite, on présentera toujours un problème de décision (i.e. un problème de la forme "w est il dans le langage L"?) sous la forme: donnée, question.

9.6 Problèmes $\mathcal{NP}$-complets

Le problème SAT

Commençons par rappeler des notions élémentaires de logique.

- Soient $u_1, u_2, \cdots u_m$ des variables propositionnelles (booleennes). Un littéral x_i est soit une variable u_j, soit une négation de variable $\overline{u_j}$.

- Une clause c_i est un ensemble de littéraux $\{x_1, x_2, ..., x_k\}$ (que l'on lit comme une disjonction).

- Une clause C est *satisfaite* par une interprétation I (qui n'est rien d'autre qu' une assignation de valeurs de vérité aux variables $u_1, u_2, ..., u_m$) si au moins un des littéraux de C est *vrai* pour I.

- Un ensemble de clauses $\mathcal{C} = \{c_1, c_2, ..., c_n\}$ est *satisfaisable* s'il existe au moins une interprétation I qui satisfait chaque clause de $\mathcal{C}$.

Théorème 9.7 (Cook) *Le problème suivant, nommé SAT, est $\mathcal{NP}$-complet:*
donnée:

$$U = \{u_1, u_2, ..., u_m\}$$

$$\mathcal{C} = \{c_1, c_2, ..., c_n\}$$

question: *Est-ce que $\mathcal{C}$ est satisfaisable?*

Idée de la preuve:

1. $SAT \in \mathcal{NP}$: on commence par parcourir la liste des variables en choisissant de façon non-déterministe, une valeur de vérité. L'application (ensemble de paires (u_i, valeur de vérité de u_i))ainsi calculée est copiée sur un deuxième ruban. Ensuite, on parcourt $\mathcal{C}$ en remplaçant les u_i par leurs valeurs (ceci s'effectue en temps $O(|\mathcal{C}| \times |\mathcal{U}|)$). Enfin, on évalue le résultat (ceci ce fait en temps $O(|\mathcal{C}|)$).

2. Pour tout $L \in \mathcal{NP}$, $L \leq_{\mathcal{P}} SAT$: Puisque $L \in \mathcal{NP}$, il existe une machine non-déterministe M qui reconnaît les éléments de L en temps $p(n)$ pour un certain polynôme p. Pour chaque donnée x, on construit un ensemble de clauses $\mathcal{C}_{M,x}$ décrivant le comportement de M qui sera satisfaisable si et seulement si il existe des choix non-déterministes faisant accepter x par M en temps $p(|x|)$. On peut supposer tout d'abord que la machine M ne comporte qu'un seul ruban. On choisit pour variables propositionnelles :

    - pour chaque état q de M et pour chaque $i \leq p(|x|)$, une variable $Q_{i,q}$ qui exprimera qu'à l'instant i, la machine M est dans l'état q

- pour chaque $1 \leq i \leq p(|x|)$, et chaque $-p(|x|) \leq c \leq p(|x|)+1$ une variable $H_{i,c}$ qui exprime qu'à l'instant i, M pointe sur la case c
- pour chaque $1 \leq i \leq p(|x|)$, pour chaque $-p(|x|) \leq c \leq p(|x|)+1$, pour chaque symbole s de l'alphabet de travail de M, une variable $S_{i,c,s}$ qui exprime que le contenu de la case c à l'instant i est s.

On forme ensuite des clauses (dont la description est donnée ci-dessous par groupes) qui expriment d'une part que l'on modélise bien une machine de Turing, d'autre part la relation de transition est celle de M.

(a) Le premier groupe de clauses exprime qu'à tout instant i, la machine M est dans exactement un état

(b) A chaque instant, la tête de lecture pointe sur exactement une case

(c) A chaque instant, chaque case contient exactement un symbole

(d) A l'instant initial, la machine est dans la configuration initiale

(e) A l'instant $p(|x|)$, la machine M est dans un état final

(f) A chaque instant i, la configuration de la machine à l'instant $i+1$ est obtenue par application d'une règle de transition de la machine M à partir de la configuration à l'instant i.

$\mathcal{C}_{M,x}$ est la réunion des 6 ensembles ci-dessus. On vérifie qu'il peut être calculé en temps polynômial par rapport à $|x|$.
■

Remarque: dans l'énoncé du théorème, la taille des données est $|\mathcal{C}| + |\mathcal{U}|$. On a en fait les inégalités suivantes: $|\mathcal{C}| \leq 2^{2|U|}$ et $|U| \leq |\mathcal{C}|$ (car les variables de U peuvent toujours être supposées apparaitre dans $\mathcal{C}$). Le théorème de Cook pourrait donc s'énoncer avec la seule donnée $\mathcal{C}$ (et pas U).

On va maintenant démontrer qu'un certain nombre d'autres problèmes sont $\mathcal{NP}$-complets en utilisant le fait que SAT l'est et le lemme 9.6.

Le problème 3SAT

Le problème nommé $3SAT$ est identique à SAT sauf que chaque clause contient exactement 3 littéraux. Il s'avère souvent utile de travailler avec $3SAT$ plutôt que SAT car sa structure est plus simple.

Théorème 9.8 *Le problème* $3SAT$ *est* $\mathcal{NP}$*-complet.*

Preuve:

1. $3SAT \in \mathcal{NP}$ car $SAT \in \mathcal{NP}$.

2. $SAT \leq_{\mathcal{P}} 3SAT$:
   Soient U et $\mathcal{C}$ les composants d'une donnée du problème SAT. Il s'agit de trouver une transformation f de U et $\mathcal{C}$ en une donnée U' et $\mathcal{C}'$ de $3SAT$ telle que :

   (a) $\mathcal{C}$ est satisfaisable si et seulement si $\mathcal{C}'$ est satisfaisable.

   (b) La transformation f est calculable en temps polynômial par rapport à la taille de la donnée $U, \mathcal{C}$. Ici, la taille est constituée par le nombre d'élements de U plus celui de $\mathcal{C}$.

   Posons $U = \{u_1, u_2, ..., u_m\}$ et $\mathcal{C} = \{c_1, c_2, ..., c_n\}$. Pour chaque clause c_i de $\mathcal{C}$, on construit un ensemble de clauses C'_i correspondant qui utilise les variables de c_i et un ensemble U'_i de nouvelles variables. La structure de C'_i dépend du nombre des littéraux de c_i :

   - $C'_i = \{\{x_i, y_i^1, y_i^2\}, \{x_i, \overline{y_i^1}, y_i^2\}, \{x_i, y_i^1, \overline{y_i^2}\}, \{x_i, \overline{y_i^1}, \overline{y_i^2}\}\}$ et $U'_i = \{y_i^1, y_i^2\}$ si $c_i = \{x_i\}$
   - $C'_i = \{\{x_i^1, x_i^2, y_i\}, \{x_i^1, x_i^2, \overline{y_i}\}\}$ et $U'_i = \{y_i\}$ si $c_i = \{x_i^1, x_i^2\}$
   - $C'_i = c_i$ et $U'_i = \emptyset$ si $c_i = \{x_i^1, x_i^2, x_i^3\}$
   - $C'_i = \{\{x_i^1, x_i^2, y_i^1\}, \{\overline{y_i^1}, x_i^3, y_i^2\}, ..., \{\overline{y_i^{k-4}}, x_i^{k-2}, y_i^{k-3}\}, \{\overline{y_i^{k-3}}, x_i^{k-1}, x_i^k\}\}$ et $U'_i = \{y_i^1, y_i^2, ..., y_i^{k-3}\}$ si $c_i = \{x_i^1, x_i^2, ..., x_i^k\}$ pour $k > 3$

   Remarquons que cette dernière construction ne serait pas possible si l'on ne s'autorisait que 2 littéraux par clause. (D'ailleurs, $2SAT \in \mathcal{P}$, comme vu en TD).

On pose finalement,

$$U' = U \cup \bigcup_{i=1}^{n} U'_i \text{ et } \mathcal{C}' = \bigcup_{i=1}^{n} C'_i.$$

Pour montrer que $\mathcal{C}$ est satisfaisable si et seulement si $\mathcal{C}'$ est satisfaisable, il suffit de montrer que C'_i est satisfaisable si et seulement si c_i est satisfaisable.

Montrons d'abord que toute interprétation I qui satisfait c_i peut être prolongée à une interprétation I' (donnant un valeur aux nouvelles variables) qui satisfait toute clause de C'_i.

- Si c_i a un, deux ou trois littéraux, peu importe la valeur des nouvelles variables : toute I' qui prolonge I satisfait chaque clause de C'_i.
- Le cas où c_i a quatre littéraux est plus délicat. Puisque I satisfait c_i, il existe au moins un des littéraux de c_i, disons x_i^l, tel que $I(x_i^l) =$ Vrai. Il suffit de prendre $I'(y_i^j) =$ Vrai si $1 \leq j \leq l-2$ et $I'(y_i^j) =$ Faux si $l-1 \leq j \leq k$

Pour la réciproque, soit I une interprétation qui satisfait toutes les clauses de C'_i. Si I satisfait au moins une des vieilles variables (celles de U), alors évidemment I satisfait c_i. D'autre coté, il est facile de voir que toute I satisfaisant toutes les clauses de C'_i forcement satisfait au moins une des vieilles variables. Il en suit que toute I satisfaisant C'_i doit satisfaire c_i.

Il nous reste à montrer que la transformation f de de U et $\mathcal{C}$ en U' et $\mathcal{C}'$ se fait en un temps borné par un polynôme de la taille de C, U. Par construction, f est en fait ici linéaire.

■

Marriage à trois

Le problème suivant est un autre exemple bien connu de problème $\mathcal{NP}$-complet.

donnée: Un ensemble $M \subseteq H \times F \times A$, où H, F, A sont trois ensembles disjoints de la même cardinalité q.
question: Existe-t-il un *mariage à trois* compatible avec M i.e. un $M' \subseteq M$ de cardinal q tel que, pour tout couple d'éléments (x, y, z), (x', y', z') de M' on a $x \neq x' \wedge y \neq y' \wedge z \neq z'$?

Le problème ci-dessus, connu aussi comme problème $3DM$ (*3-dimensional matching*), est une généralisation du problème suivant, dit du *mariage*. Soient H un ensemble de q hommes, F un ensemble de q femmes et $M \subseteq H \times F$ tel que $(h, f) \in M$ indique que h et f n'ont rien contre l'idée de s'épouser. La question est de savoir s'il est possible de combiner des mariages tels que :

1. tout le monde est marié
2. il n'y a pas de polygamie
3. si h épouse f, alors h et f n'avaient rien contre l'idée de s'épouser,

Remarquons que ce problème classique est dans la classe $\mathcal{P}$.

Le problème $3DM$ généralise le problème classique du mariage en ce sens qu'une donnée de $3DM$ comporte "trois sexes": H, F et A.

La preuve du fait que $3DM$ est dans $\mathcal{NP}$ est laissée en exercice. La preuve du fait que $3DM$ est $\mathcal{NP}$ difficile consiste à montrer que $3SAT \leq_P 3DM$.

Soit donc

$$U = \{u_1, u_2, ..., u_n\}$$

$$\mathcal{C} = \{c_1, c_2, ..., c_m\}$$

une donnée de $3SAT$; on lui associe la donnée de $3DM$ suivante.

- $H = \{u_{ij}, \overline{u}_{ij} : 1 \leq i \leq n, 1 \leq j \leq m\}$
- $F = F_1 \cup F_2 \cup F_3$ où
  - $F_1 = \{f_{ij} : 1 \leq i \leq n, 1 \leq j \leq m\}$
  - $F_2 = \{s_j^1 : 1 \leq j \leq m\}$
  - $F_3 = \{g_j^1 : 1 \leq j \leq m(n-1)\}$
- $A = A_1 \cup A_2 \cup A_3$ où

- $A_1 = \{a_{ij} : 1 \leq i \leq n, 1 \leq j \leq m\}$
- $A_2 = \{s_j^2 : 1 \leq j \leq m\}$
- $A_3 = \{g_j^2 : 1 \leq j \leq m(n-1)\}$

Remarquons que $\mid H \mid = \mid F \mid = \mid A \mid = 2mn$. Comme on verra, les ensembles H, F_1, F_2, A_1 et A_2 jouent un rôle fondamental dans notre construction; par contre, F_3 et A_3 sont là exclusivement afin d'avoir la même cardinalité q pour les ensembles H, F, A.

Il nous reste à définir M. Cet ensemble de triplets est l'union de trois groupes de triplets, M_1, M_2 et M_3.

- Le premier ensemble de mariages possibles a pour objectif de contraindre les m exemplaires $u_{i1}, \ldots, u_{im}$ de la variable propositionnelle u_i à avoir la même interprétation. $M_1 = \bigcup_{u_i \in U} INT_i$ où $INT_i = V_i \cup F_i$ et

  - $V_i = \{(\overline{u_{ij}}, f_{ij}, a_{ij}) \ : \ 1 \leq j \leq m\}$
  - $F_i = \{(u_{ij}, f_{i,j+1}, a_{ij}) \ : \ 1 \leq j < m\} \cup \{(u_{im}, f_{i1}, a_{im})\}$

  Remarque 9.1 *Observons que, s'il existe un mariage compatible avec M_1, alors, pour chaque i, ce mariage contiendra ou bien tous les triplets de V_i ou bien tous les triplets de F_i, ce qui correspond respectivement à l'assignation de u_i à vrai ou à faux. En effet, si $INT_i' \subseteq INT_i$ contient m triplets dont au moins un de V_i et un dans F_i, alors INT_i' contiendra deux triplets ayant une composante f_{ij} ou une composante a_{ij} en commun (voir figure 16).*

- Le deuxième ensemble de mariages possibles sert à assurer que chaque clause est satisfaite.

  $$M_2 = \bigcup_{c_j \in \mathcal{C}} C_j$$

  où

  $$C_j = \{(u_{ij}, s_j^1, s_j^2) \ : \ u_i \in c_j\} \cup \{(\overline{u_{ij}}, s_j^1, s_j^2) \ : \ \overline{u_i} \in c_j\}$$

  Remarque 9.2 *Pour obtenir un mariage acceptable, il n'est pas possible de choisir 2 triplets dans C_j, car les deuxièmes et troisièmes composantes des triplets de C_j sont identiques.*

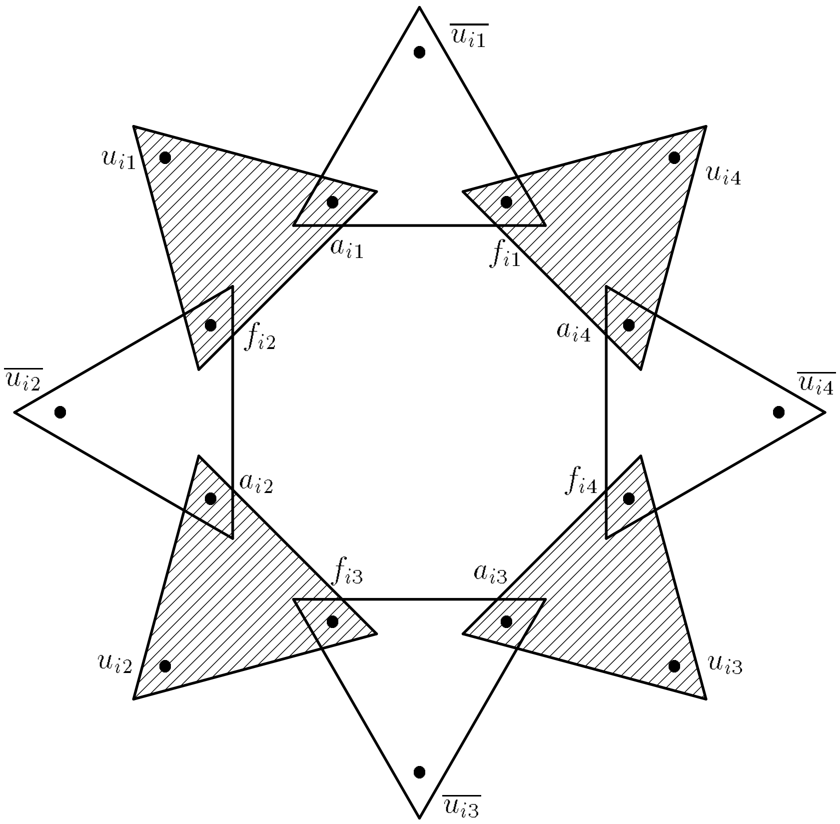

Figure 16: Les interprétations des u_{ij} sont forcées être indépendantes de j

- Le dernier ensemble de mariages possibles est défini par:

  $M_3 = \{(u_{ij}, g_k^1, g_k^2), (\overline{u_{ij}}, g_k^1, g_k^2) \ : \ 1 \leq k \leq m \times (m-1), 1 \leq i \leq n, 1 \leq j \leq m\}$

Montrons que $3DM$ a une solution si et seulement si $3SAT$ a une solution.

Si $3DM$ a une solution, alors $3SAT$ a aussi une solution

Soit M' une solution pour $3DM$, i.e. un mariage compatible avec M. Alors, $M' \subseteq M$, $|M'| = 2 \times m \times n$ et les triplets de M' sont tous différents, composante par composante.

- Pour tout $1 \leq i \leq n$, M' contient nécessairement m triplets de INT_i. En effet, si ce n'était pas le cas, il existerait une femme f_{ij} ou bien un ange a_{ij} qui ne serait pas marié.
- Par la remarque 9.1, ces m triplets sont ou bien tous dans V_i, ou bien tous dans F_i.

Définissons alors une interprétation I :

$$I(u_i) = \begin{cases} \text{Vrai} & \text{Si} \quad M' \cap Int_i = V_i \\ \text{Faux} & \text{Si} \quad M' \cap Int_i = F_i \end{cases}$$

Remarque 9.3 *Observons que $I(u_i) =$ Vrai si et seulement si aucun triplet de la forme (u_{ij}, x, y) n'appartient à $M' \cap INT_i$.*

Montrons que I satisfait bien toutes les clauses de $\mathcal{C}$. Pour tout $1 \leq j \leq m$, M' contient exactement un triplet de C_j. En effet, il en contient au moins un sinon, ni la femme s_j^1, ni l'ange s_j^2 ne seraient mariés. Il n'en contient pas plus d'un par la remarque 9.2.

Soit par exemple (u_{ij}, s_j^1, s_j^2) ce triplet (le cas $(\overline{u_{ij}}, s_j^1, s_j^2)$ est symétrique). Par définition de C_j, u_{ij} est un litéral de la clause c_j. Par ailleurs, aucun triplet de $M' \cap INT_i$ n'est de la forme (u_{ij}, x, y). Par la remarque 9.3, la clause c_{ij} contient donc un litéral u_{ij} tel que $I(u_{ij}) =$ Vrai.

Si $3SAT$ a une solution, alors $3DM$ a une solution

Soit I une interprétation qui satisfait toutes les clauses de $\mathcal{C}$. On définit M' de la façon suivante:

- Si $I(u_i) = \text{Vrai}$, on marie $\overline{u_{i1}}, \ldots, \overline{u_{im}}$ comme indiqué par V_i: $V_i \subseteq M'$
- Inversement, si $I(u_i) = \text{Faux}$, $F_i \subseteq M'$.
- Pour tout j, soit u_i (resp. $\overline{u_i}$) un litéral de c_j tel que $I(u_i) = \text{Vrai}$ (resp. $I(u_i) = \text{Faux}$). Par symétrie, nous supposerons désormais sans perdre de généralité que le litéral choisi est u_i. On marie u_{ij} à s_j^1 et s_j^2: $(u_{ij}, s_j^1, s_j^2) \in M'$.

Nous avons ainsi marié $n \times m + m$ hommes. Il en reste donc $m \times (n-1)$ à caser. Pour ceux-là, on peut utiliser la composante M_3 qui nous assure qu'il reste exactement $m \times (n-1)$ anges libres et exactement $m \times (n-1)$ femmes libres.

On vérifie par ailleurs que la transformation est bien polynômiale. ■

9.7 Problèmes $\mathcal{P}$Espace-complets

Definition 9.1 *Un langage L est* $\mathcal{P}$ESPACE-complet *si les 2 conditions suivantes sont satisfaites:*

1. $L \in \mathcal{P}\text{ESPACE}$
2. *pour tout langage $L' \in \mathcal{P}\text{ESPACE}$, $L' \leq_{\mathcal{P}} L$ (L' est $\mathcal{P}$ESPACE-difficile).*

Remarquons que, si $\mathcal{P} \neq \mathcal{P}\text{ESPACE}$, alors un problème $\mathcal{P}$ESPACE-complet ne peut pas être dans $\mathcal{P}$.

Un problème $\mathcal{P}$ESPACE-complet typique est le problème des formules Booléennes quantifiées (QBF en anglais) dont voici l'énoncé:

Donnée : une formule du calcul propositionnel, précédée de quantifications sur les variables propositionnelles:

$$F = Q_1 x_1 \ldots Q_n x_n \quad \Phi$$

où Φ est une formule du calcul propositionnel en forme normale conjonctive et, pour tout i, Q_i est $\exists$ ou $\forall$.

Question : F est-elle vraie ?

Théorème 9.9 *QBF est* $\mathcal{P}$ESPACE*-complet*

Idée de la preuve

1. Pour montrer que QBF $\in$ $\mathcal{P}$ESPACE, on procède comme pour SAT: On considère successivement toutes les assignations des variables propositionnelles. Un ruban de la machine permet de garder trace des assignations déjà considérées (en espace linéaire); par exemple, si l'on numérote les assignations possibles de 1 à 2^n, on écrit le numéro de l'assignation courante en base 2. Enfin, le calcul de la valeur de vérité d'une formule propositionnelle en forme conjonctive, l'assignation étant donnée, se fait en temps polynômial comme déjà vu.

2. Il est beaucoup plus délicat de montrer que tout problème $\mathcal{P}$ESPACE se réduit en temps polynômial à QBF. L'idée principale est de considérer une variable propositionnelle par case du ruban. Une assignation à une suite de n variables propositionnelles représente un mot sur l'alphabet $\{0, 1\}$. Les quantifications sur les variables propositionelles permettent alors d'exprimer des quantifications sur les configurations. On écrit alors une formule:

$$\exists \vec{P} \exists \vec{Q}.\ (\text{Initial}(\vec{P}) \wedge \text{Accepte}(\vec{Q}) \wedge \text{Accessible}_{c^n}(\vec{P}, \vec{Q}))$$

   qui sera vraie si et seulement s'il existe des assignations de $\vec{P}$ et $\vec{Q}$ qui codent respectivement des configurations initiale et finale et telles que l'on puisse accéder de $\vec{P}$ à $\vec{Q}$ en au plus c^n étapes de calcul. D'autre part, il a été vu (théorème 8.9) qu'il existe une constante c_M telle que si M accepte x en espace $p(|x|)$, alors M accepte x en temps au plus $c^{p(|x|)}$. Il ne reste donc plus qu'à montrer que, si $L(M) \in$ $\mathcal{P}$ESPACE, alors les formules "Initiale", "Accepte", "Accessible$_{c^n}$" peuvent être construites en temps polynômial par rapport à n. Pour la construction de "Accessible$_{c^n}$", on utilise une méthode analogue à celle qui est utilisée dans la preuve du théorème de Savitch. (Mais il y a une petite astuce supplémentaire). ■

De nombreux problèmes de jeux peuvent être prouvés $\mathcal{P}$ESPACE-complets en réduisant QBF. (Par exemple, le problème de stratégie gagnante pour le jeu de Hex, ou le problème de géographie généralisée vu en TD). Un autre exemple de problème $\mathcal{P}$ESPACE-complet est le problème suivant: étant donnée une expression régulière e (sur un alphabet Σ) est-ce que e représente le langage Σ^*. (En fait, l'équivalence des expressions régulières est $\mathcal{P}$ESPACE-complete).

9.8 Autres classes de complexité

Soit L un langage. $co(L)$ est une autre notation pour $\overline{L}$.

Définition 9.10 $co(\mathcal{P}) = \{L | co(L) \in \mathcal{P}\}$ *et* $co(\mathcal{NP}) = \{L | co(L) \in \mathcal{NP}\}$

Il est facile de voir que $\mathcal{P} = co(\mathcal{P})$ mais que dire de $\mathcal{NP} = co(\mathcal{NP})$? On peut conjecturer que cette égalité est fausse; par exemple on ne voit pas comment prouver, même avec un algorithme non déterministe, qu'un ensemble de clauses est insatisfaisable sans tester *toutes* les 2^n interprétations possibles. Toutefois nous sommes à nouveau en presence d'un problème ouvert, car la question

$$co(\mathcal{P}) \stackrel{?}{=} co(\mathcal{NP})$$

n'a pas été résolue.

Aucune des classes de complexité que l'on a consideré jusqu' à ici ne contient de problème de décision pour lesquels on a *démontré* qu'il n'existe pas d'algorithmes déterministes polynômiaux; par exemple, on dit que SAT n'est pas "faisable", mais ceci car on "croit" que $\mathcal{P} \neq \mathcal{NP}$.... Il existe toutefois des problèmes qui ont une complexité intrinsèquement exponentielle: ceux-ci sont *prouvés* "infaisables". ce qui motive la définition suivante.

Définition 9.11

$$\mathcal{EXP}\text{TEMPS} = \bigcup_{i>0} \text{DTEMPS}(2^{n^i})$$

$$\mathcal{NEXP}\text{TEMPS} = \bigcup_{i>0} \text{NTEMPS}(2^{n^i})$$

Par exemple, le problème suivant peut-être prouvé $\mathcal{NEXP}$TEMPS-complet:

Donnée : deux expressions régulières e_1 et e_2 construites à l'aide de $\{\cup, \cdot, {}^2\}$ (union, concaténation, carré, et les constantes 0,1)

Question : e_1 et e_2 définissent-elles deux langages distincts ?

La classe $\mathcal{NP}$ se trouve être *strictement* inclue dans la classe $\mathcal{NEXP}$Temps. Donc même si l'on avait $\mathcal{P} = \mathcal{NP}$, il existerait quand même des problèmes décidables mais qui ne peuvent être résolus par aucun algorithme polynômial.