1 Introduction à la calculabilité

La calculabilité est une branche de l'informatique-mathématique qui a pour but de déterminer si un problème donné a une solution algorithmique ou non. L'idée peut sembler étrange a première vue; existe-t-il vraiment des problèmes que l'on puisse énoncer pour lesquels on ne puisse fournir de solution algorithmique?

Considérons un exemple dans le langage PASCAL. Cet exemple que l'on nomme le **problème d'arrêt** est le suivant:

Exemple 1.1 *Peut-on écrire dans le langage PASCAL une fonction* $Arret(F, D{:}Text){:}Boolean$; *qui décide si une fonction* $Fun(D{:}Text){:}Boolean$; *donnée en PASCAL dans le fichier F, s'arrêtera ou bouclera à l'infini sur une donnée D?*

On montre par contradiction qu'Arret ne peut exister. Supposons au contraire qu'une telle fonction existe, alors on peut l'écrire:

```
FUNCTION Arret(F,D:Text):Boolean;
{================================}

{ SI     "F contient une fonction PASCAL correcte
          nomme Fun, a un argument D de type Text
          et que Fun(D) ne boucle pas a l'infini"
  ALORS Arret:=True
  SINON Arret:=False }

BEGIN
  ...
  code de la fonction Arret
  ...
END;
```

Dans un fichier de type Text nommé FunText on met la fonction PASCAL suivante:

```
FUNCTION Fun(D:Text):Boolean;
{===========================}

    FUNCTION Arret(F,D:Text):Boolean;
    {================================}

    BEGIN
      ...
      code de la fonction Arret
      ...
    END;

BEGIN
    IF Arret(D,D) THEN WHILE True DO Fun:=False {boucle infinie}
                  ELSE Fun:=True
END;
```

Que vaut $Arret(FunText, FunText)$*?*

Notons d'abord que sur donnée D la fonction Fun bouclera à l'infini si $Arret(D,D)$ *retourne True, c'est-à-dire si D contient une fonction PASCAL correcte et que cette fonction s'arrête sur donnée D. Notons ensuite que* $FunText$ *contient le code d'une fonction PASCAL correcte (décrite plus haut). Supposons que* $Arret(FunText, FunText) = True$*, par les deux observations que l'on vient de faire* $Fun(FunText)$ *boucle à l'infini. Mais puisque* $FunText$ *contient la fonction* Fun*, ceci implique que* $Arret(FunText, FunText) = False$ *(ce qui est une contradiction). Si on fait l'hypothèse inverse, que* $Arret(FunText, FunText) = False$ *on en déduit que* $Arret(FunText, FunText) = True$*. Il en résulte donc que la fonction Arret* ***ne peut accomplir sa tâche*** *correctement, donc qu'une telle fonction ne peut être écrite en PASCAL.*
■

L'exemple suivant est encore plus déroutant.

Exemple 1.2 *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. Par exemple*

```
FUNCTION Beaucoup:Integer;
{=======================}

CONST X=100000;

BEGIN
    Beaucoup:=X*X*X*X*X*X*X*X
END;
```

fait partie de $PASC_0^{112}$ car elle est composée de 112 caractères (en comptant les " " et les fin-de-lignes) et n'a pas de paramètre d'entrée. Considérons la fonction (mathématique) $SUP(n)$ 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

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

Par exemple

$$SUP(112) \geq 1000000000000000000000000000000000000000$$

car cette valeur est retourné par la fonction Beaucoup, laquelle est dans $PASC_0^{112}$.

Peut-on écrire en PASCAL une fonction Super(n:Integer):Integer telle que pour tout n, $Super(n) = SUP(n)$?

En fait, non seulement ne peut-on écrire SUP en PASCAL mais de plus pour toute fonction $Super(n)$ que l'on peut écrire en PASCAL, $SUP(n)$ borne supérieurement $Super(n)$ pour tout n assez grand. C'est-à-dire

$$\forall Super \in PASC_1 \exists n_0 \forall n > n_0 \ \ [SUP(n) > Super(n)]$$

Preuve

On construit la fonction PASCAL suivante:

```
FUNCTION Geant:Integer;
CONST N0=...;X=0;
FUNCTION Super(n:Integer):Integer;
...
BEGIN
 ...
 code de la fonction Super
 ...
END;
BEGIN Geant:=Super(N0)+1 END;
```

Quelle est la longueur de la fonction Geant?
Cela dépend de la constante que l'on associe à $N0$.

$$\mid Geant \mid = 67 + \mid Super \mid + \mid N0 \mid$$

On veut ajuster $N0$ de façon à ce que $N0 = \mid Geant \mid$. Pour ceci on assigne à $N0$ une certaine valeur (approximativement $N0 = 67 + \mid Super \mid + \log_{10}(67 + \mid Super \mid)$) de façon à ce que

$$0 \leq N0 - \mid Geant \mid \leq 1.$$

X sert à ajuster la longueur de Geant de façon à ce que $N0$ soit exactement $\mid Geant \mid$. Si $N0 = \mid Geant \mid$ alors on laisse $X = 0$ sinon on met $X = 10$ (ce qui va forcer $N0 = \mid Geant \mid$).

Le resultat est que Geant sera une fonction de $N0$ caractères qui retournera $Super(N0) + 1$, donc

$$Super(N0) \neq \max_{val(f)}\{f \in PASC_0^{N0}\}$$

car $Geant \in PASC_0^{N0}$ et $val(Geant) > Super(N0)$.

On obtient le même résultat pour tout $N > N0$ en créant sucèssivement des fonctions Geant de longueur $N0, N0+1, N0+2, \ldots..$ Pour passer de $N0$

à $N0+1$ on ajoute 1 à la constante associée à $N0$ et si ceci ne change pas la longueur de Geant on rajoute un chiffre à la constante X. On aura de nouveau que $N0 = \mid Geant \mid$, mais avec un caractère de plus. ■

Est-ce que cette situation provient de la "faiblesse" expressive du langage PASCAL ou est-ce que tout langage de programmation aurait les même problèmes? En fait, peux importe le langage dans lequel on exprime nos algorithmes, il y aura toujours des fonctions qui ne seront pas **calculables**. Pour étudier ceci nous allons devoir formaliser mathématiquement la notion de **fonctions calculables**. On verra succèssivement trois formalismes différents, certain étant plutôt "mathématiques", d'autres étant plutôt "informatiques", qui nous mèneront à des notions équivalentes de calculabilité. On verra d'abord les fonctions récursives, ensuite les fonctions calculables par Machines de Turing, puis enfin les fonctions définissables par Lambda calcul.

2 Fonctions récursives

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 partielles récursives*. Ce premier ensemble est définit par induction à partir de l'ensemble suivant de fonctions initiales.

Définition 2.1 *Les fonctions initiales sont U_i^k, S et Z où*

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

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

On définit d'abord les opérations de composition, récursion primitive et minimalisation 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 pour $\chi, \psi_1, \psi_2, ..., \psi_m \in F$*

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

Définition 2.3 *F est fermé par* <u>*récursion primitive*</u> *si pour $\chi, \psi \in F$*

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

est aussi $\in F$.

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

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

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 minimalisation.*

Autrement dit, les fonctions récursives sont toutes les fonctions que l'on peut construire à partir des fonctions initiales par composition, récursion primitive et minimalisation. Toutes les fonctions avec lesquelles nous sommes familier, 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 $\chi(x) = x = U_1^1(x)$ et $\psi(x, y, z) = x + 1 = S(U_1^3(x, y, z))$. Puisque U_1^1, U_1^3 et S sont initiales donc récursives, ψ est récursive par composition et f par récursion primitive.

2.1 Fonctions partielles récursives

On définit maintenant les opérations de composition et minimalisation sur un ensemble de fonctions partielles afin de définir ensuite l'ensemble des fonctions partielles récursives. On dit d'un ensemble F de fonctions partielles que

Définition 2.6 *F est fermé par composition si pour $\chi, \psi_1, \psi_2, ..., \psi_m \in F$*

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

avec la convention que $\varphi(\vec{n}) \uparrow$ si une des $\psi_i(\vec{n}) \uparrow$ ou $\chi(\psi_1(\vec{n}), \psi_2(\vec{n}), ..., \psi_m(\vec{n})) \uparrow$.

Définition 2.7 *F est fermé par minimalisation-partielle 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 2.8 *l'ensemble $\wp\Re$ des fonctions partielles récursives est le plus petit ensemble contenant les fonctions récursives qui soit fermé par composition et minimalisation-partielle.*

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

Exemple 2.2 *$f(a, b) \begin{cases} = 0 & si\ a = b = 0 \\ \uparrow & sinon \end{cases}$ est une fonction partielle 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 partielle récursive par minimalisation-partielle et de même f est partielle récursive par composition.

2.2 Ensembles récursifs et réc. énumérables

Soit $L \subseteq N$, un ensemble d'entier. Soit λ la fonction caractéristique de L

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

et soit φ la fonction partielle caractéristique de L

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

Définition 2.9 *L est un <u>ensemble récursif</u> si λ est une fonction récursive.*

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

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

Définition 2.10 *L est un <u>ensemble récursivement énumérable</u> (noté <u>r.é.</u>) si φ est une fonction partielle récursive.*

Exemple 2.5 *$L = \{n \mid \exists a, b, c > 0 \ [a^n + b^n = c^n]\}$ est r.é., mais n'est peut-être pas récursif.*

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.

Justification

Si l'on considère λ comme une propriété des nombres entiers (n possède la propriété si et seulement si $\lambda(n) = 1$) alors

Théorème 2.11 *L est r.é. si pour tout $n \in L$ il existe une preuve (dérivable à partir des axiomes de l'arithmétique) que $\lambda(n) = 1$.*

De même,

Théorème 2.12 *L est récursif si L est r.é. et de plus pour tout $n \notin L$ $\lambda(n) = 0$ est prouvable (à partir des axiomes de l'arithmétique).*

Ce fait est la motivation de base du développement de la théorie des fonctions récursives.

3 les Machines de Turing

Bien que la théorie des fonctions récursives nous fournisse une définition juste de la notion de calculabilité, il n'en reste pas moins que notre intuition reste sur sa faim. Les notions de machines, d'algorithmes et de langages de programmation n'interviennent même pas dans cet approche. La machine de Turing est une modélisation mathématique du fonctionnement d'un ordinateur très simple mais quand même très puissant. Ce modèle développé par le mathématicien Alan Turing ressemble essentiellement à ce que l'on voit à la figure 1.

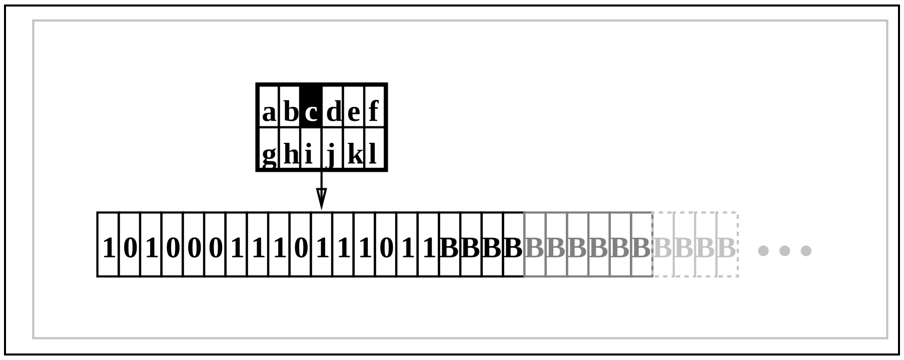

Figure 1: Machine de Turing

La machine travaille sur un ruban infini 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 détat, 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 *Formellement, la 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'ensemble des symboles sur lesquelles la machine travaille (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 fournies en entrée à la machine sont exprimés, q_0 est l'élement de Q dans lequel la machine se trouve au début de son calcul (on dit* état initial*), $F \subseteq Q$ est l'ensemble des états acceptants (états dans lesquels l'arrêt de la machine définie l'acceptation du mot fournie en entrée sur le ruban de travail) et $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{\triangleleft, \bigtriangledown, \triangleright\}$ est la fonction de transition permettant à la machine de transformer son état, de modifier le symbol qu'elle voit présentement et de se déplacer vers la gauche ($\triangleleft$), vers la droite ($\triangleright$) 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 $Q \times \Gamma$.*

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 et sa tête de lecture sur la première case du ruban de travail. Un mot $\omega \in \Sigma^*$ 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 lequelle 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 pair (q, σ) où δ n'est pas définie. La machine s'arrête alors.

Définition 3.2 *Une* description instantanée *(DI) de la configuration d'une machine M est dénotée par $\alpha_1 q \alpha_2$ avec $q \in Q$ et $\alpha_1, \alpha_2 \in \Gamma^*$. Cette représentation veut dire que M est présentement dans l'état q et que 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.*

On définit une *transition* de M à partir d'une DI donnée $x_1x_2...x_{i-1}qx_i...x_n$.

Définition 3.3 *Supposons que $\delta(q, x_i) = (q', y, \triangleleft)$ (si $n = i - 1$ alors on prends $x_i = B$). On écrit alors*

$$x_1x_2...x_{i-1}qx_i...x_n \vdash_M x_1x_2...x_{i-2}q'x_{i-1}yx_{i+1}...x_n$$

pour exprimer le fait que M est passée à une nouvelle configuration par cette transition (notons que si $i = 1$ aucune DI ne peut succéder à $qx_1...x_n$ par *la relation $\vdash_M$ car la tête de lecture ne peut "quiter" le ruban). De même si $\delta(q, x_i) = (q', y, \triangleright)$ on écrit*

$$x_1x_2...x_{i-1}qx_i...x_n \vdash_M x_1x_2...x_{i-1}yq'x_{i+1}...x_n$$

et si $\delta(q, x_i) = (q', y, \bigtriangledown)$ on écrit

$$x_1x_2...x_{i-1}qx_i...x_n \vdash_M x_1x_2...x_{i-1}q'yx_{i+1}...x_n.$$

Quand on a $\alpha_1q\alpha_2 \vdash_M \alpha_3q'\alpha_4$ on dit que la seconde DI découle de la première en une transition. Quand on a $\alpha_1q\alpha_2 \vdash_M \alpha_3q'\alpha_4 \vdash_M ... \vdash_M \alpha_ip\alpha_{i+1}$ on écrit

$$\alpha_1q\alpha_2 \vdash_M^* \alpha_ip\alpha_{i+1}$$

et on dit que $\alpha_iq\alpha_{i+1}$ découle $\alpha_1q\alpha_2$.

Définition 3.4 *Le langage accepté par une machine de Turing M est:*

$$L(M) = \{\omega \mid \omega \in \Sigma^* \text{ et } q_0\omega \vdash_M^* \alpha_1q\alpha_2 \text{ pour un } q \in F \text{ et des } \alpha_1, \alpha_2 \in \Gamma^*\}$$

Par convention, on écrit toujours δ de façon à ce qu'elle ne soit jamais définit 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 $\{0^n1^n \mid n > 0\}$.*

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

où δ est donnée sous forme de table à la figure 2. La machine fonctionne en

$Q \times \Gamma$	0	1	X	Y	B
q_0	$(q_1, X, \triangleright)$	-	-	$(q_3, Y, \triangleright)$	-
q_1	$(q_1, 0, \triangleright)$	$(q_2, Y, \triangleleft)$	-	$(q_1, Y, \triangleright)$	-
q_2	$(q_2, 0, \triangleleft)$	-	$(q_0, X, \triangleright)$	$(q_2, Y, \triangleleft)$	-
q_3	-	-	-	$(q_3, Y, \triangleright)$	$(q_4, B, \triangleright)$
q_4	-	-	-	-	-

Figure 2: *la fonction δ*

remplaçant succèsivement 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 rejète le mot fournie en entrée. Voici un exemple du calcul de M sur donnée $\omega = 0011$:

$q_0 0011$	$\vdash Xq_1 011$	$\vdash X0q_1 11$	$\vdash Xq_2 0Y1$	
	$\vdash q_2 X0Y1$	$\vdash Xq_0 0Y1$	$\vdash XXq_1 Y1$	
	$\vdash XXYq_1 1$	$\vdash XXq_2 YY$	$\vdash Xq_2 XYY$	
	$\vdash XXq_0 YY$	$\vdash XXYq_3 Y$	$\vdash XXYYq_3$	$\vdash XXYYBq_4$

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

3.1 langages et fonctions Turing-calculables

Définition 3.5 *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 $\omega \in \Sigma^*$.*

En plus de pouvoir accepter des langages, les machines de Turing peuvent être utilisées pour calculer des fonctions (possiblement partielles) à valeur entière de plusieurs arguments de $\bigcup_{k \in N} \{f : N^k \to N\}$. 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 $0^{n_1} 1 0^{n_2} 1 ... 1 0^{n_k}$. 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 3.6 *Une fonction partielle* $f : N^k \rightarrow N$ *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* $f : N^k \rightarrow N$ *est* Turing-calculable *s'il existe une machine* M *telle que pour tout* $\vec{n}$ *on a que* $M(\vec{n}) = f(\vec{n})$.

Exemple 3.2 *Comme premier exemple prenons une machine* M *qui calcul* $n + m$ *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 calcul* $n \dot{-} m$ *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
q_0	$(q_1, B, \triangleright)$	$(q_5, B, \triangleright)$	-
q_1	$(q_1, 0, \triangleright)$	$(q_2, 1, \triangleright)$	-
q_2	$(q_3, 1, \triangleleft)$	$(q_2, 1, \triangleright)$	$(q_4, B, \triangleleft)$
q_3	$(q_3, 0, \triangleleft)$	$(q_1, 1, \triangleleft)$	$(q_0, B, \triangleright)$
q_4	$(q_4, 0, \triangleleft)$	$(q_4, B, \triangleleft)$	$(q_6, 0, \triangleright)$
q_5	$(q_5, B, \triangleright)$	$(q_5, B, \triangleright)$	$(q_6, B, \triangleright)$
q_6	-	-	-

Figure 5*: la fonction δ*

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

$q_0 0010$	$\vdash Bq_1 010$	$\vdash B0q_1 10$	$\vdash B01q_2 0$		
	$\vdash B0q_3 11$	$\vdash Bq_3 011$	$\vdash q_3 B011$		
	$\vdash Bq_0 011$	$\vdash BBq_1 11$	$\vdash BB1q_2 1$		
	$\vdash BB11q_2$	$\vdash BB1q_4 1$	$\vdash BBq_4 1$	$\vdash Bq_4$	$\vdash B0q_6$

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

$q_0 0100$	$\vdash Bq_1 100$	$\vdash B1q_2 00$	$\vdash Bq_3 110$
	$\vdash q_3 B110$	$\vdash Bq_0 110$	$\vdash BBq_5 10$
	$\vdash BBBq_5 0$	$\vdash BBBBq_5$	$\vdash BBBBBq_6$

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

3.2 construction de machine 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 symbole de Γ en augmentant son nombre d'états. Soit Q un ensemble d'états possibles d'une machine M. On cré l'ensemble Q' des super-états de M en posant $Q' = Q \times \underbrace{\Gamma \times \Gamma \times ... \times \Gamma}_{k}$. 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*

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

On construit une machine acceptant L en mémorisant le premier caractére lu sur le ruban et ensuite en le cherchant dans les 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*

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

Cette mémoire permet entre autre á 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 emagasinée sur les autres. Pour ce faire on modifie Γ afin de pouvoir conserver plusieurs symboles à la fois sur le ruban. On cré l'ensemble Γ' des supersymboles de M en posant $\Gamma' = \underbrace{\Gamma \times \Gamma \times ... \times \Gamma}_{k}$. Chaque supersymbole permet mémoriser k éléments de Γ dans une case du ruban.

Exemple 3.5 *Considérons par exemple le langage $L = \{\omega \in \{0,1\}^* \mid \omega$ est premier lorqu'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*

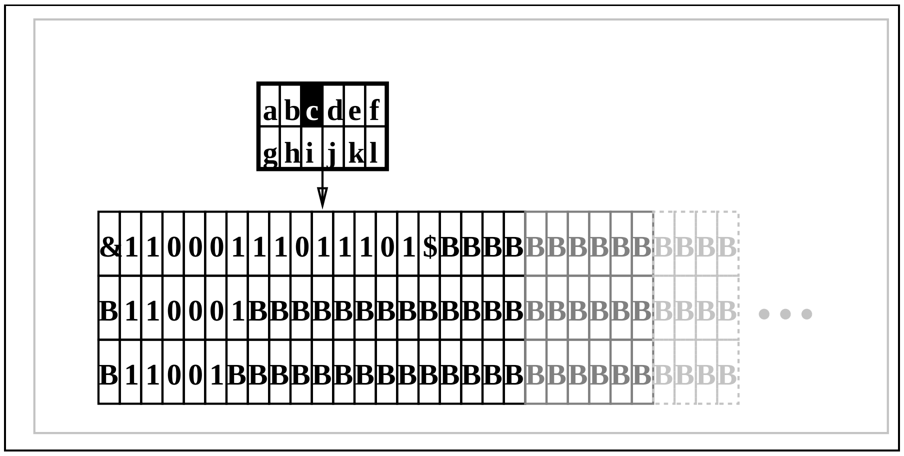

Figure 8: *Machine de Turing avec ruban 3-pistes*

que l'on transforme d'abord en &ω\$ 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 decider de la primalité du nombre en entrée est décrire sur la seconde piste les nombres binaires $2, 3, \ldots, \omega - 1$ *succèssivement et pour chacun de recopier* ω *sur la troisième piste puis de soustraire le contenu de la seconde piste d'*ω *tant que cela reste possible. Si la soustraction répétée mène à 0 alors* ω *n'est pas premier, sinon si le reste (sur la troisième piste) est non-nul pour tous les diviseurs (sur la deuxième piste), c'est que* ω *est premier. On voit un exemple de ce procédé à la figure 8 où la machine est en train de vérifier si le nombre* $\omega = 11000111011101$ *est divisible par* 1100011.

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 $\surd$. Cet outil 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 mirroir 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.*

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

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:*

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

Voici 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 á 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 model standard mais différentes de façon non-triviale. On montre que chacun de ces models est équivalent au model standard.

ruban infini bidirectionel

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

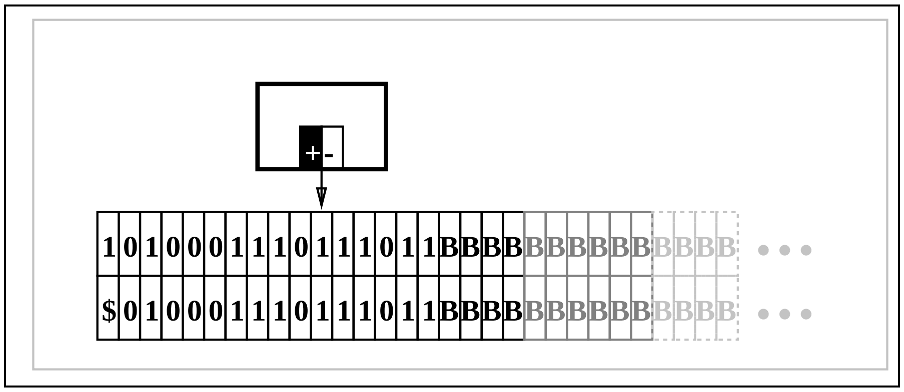

Figure 10: simulation d'un ruban infini bidirectionel

D'une part il est évident que tout calcul se faisant dans le model standard se fait aussi dans ce model puisqu'il suffit de définir une case "départ" et de ne jamais la dépasser à sa gauche. Par contre pour simuler le model bidirectionel avec le model 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 positives et sur la seconde piste les symboles en positions négatives. La machine se rappel du côté considéré au moyen de deux états "+" et "-".

plusieurs têtes et rubans

Similairement á la notion de machines multi-pistes, on peut définir un model avec plusieurs rubans disjoint 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.

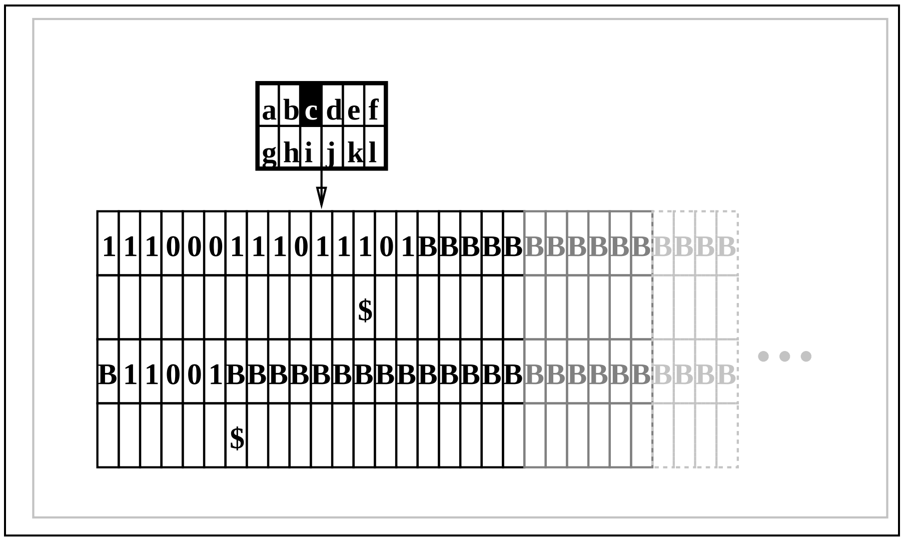

Figure 11: simulation de 2 rubans avec 2 têtes de lectures

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 multiples rubans, tandis que d'autres pistes servent à mémoriser la position des multiples têtes de lectures. Vous pouvez imaginer le reste...

Dans l'exemple suivant, on simule une machine à deux rubans et deux têtes. Le symbole "\$" sert à identifier la position des têtes de lecture.

machines non-déterministes

Une machine de Turing non-déterministe est très semblable au model standard sauf pour le fait que δ n'est plus une fonction mais simplement une relation, c'est-à-dire que pour $q \in Q$ et $\gamma \in \Gamma$ il se peut que $\delta(q, \gamma)$ ait plusieurs valeurs possibles. Pour cette raison, il se peut que pour une DI 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 cette dérivation n'est pas unique.

Théorème 3.7 *S'il existe une machine non-déterminste M acceptant un langage L alors il existe aussi une machine standard M' acceptant le même langage.*

Preuve

On cosntruit la machine M' en lui donnant trois rubans et en lui fournissant une version étendue de l'alphabet de travail Γ de M. On prends $\Gamma' = \Gamma \cup (Q \times \Gamma \times \{\lhd, \bigtriangledown, \rhd\})$ afin que M' puisse écrire sur un de ses rubans des triplets de la forme (q, γ, m) représentant les différentes possibilités de transitions de la machine M. M' a son mot d'entrée sur son premier ruban. Sur le second ruban elle génère succèssivement toutes les suites finies d'éléments de $(Q \times \Gamma \times \{\lhd, \bigtriangledown, \rhd\})$ en commençant par les séquences les plus courtes. M' possède dans sa mémoire locale finie une copie de la fonction δ de M lui permettant de vérifier lesquelles des transitions sont possibles dans le calcul non-déterministe de M.

M' copie d'abord son mot d'entrée sur son troisième ruban, puis simule le calcul de M sur cette donnée. Quand M a plusieurs choix possibles (donnés par δ), M' utilise le choix offert sur son second ruban comme valeur de δ (si cette valeur est effectivement possible par δ). Si M' "voit" que M serait alors dans un état acceptant, elle accèpte aussi. Si M' manque de choix (elle a utilisé tous les choix disponibles sur son deuxième ruban ou si le prochain choix n'est pas légal selon δ) elle arrête son calcul, passe à la prochaine séquence de choix possibles et reprend le calcul depuis le début avec ces nouveaux choix.

Si M finit par accepter sa donnée, M' finira par trouver une séquence légale de choix où elle acceptera aussi. Si M n'accepte pas l'entrée alors M' ne trouvera jamais de choix la faisant accepter non plus.
■

nombre limité de symboles ou d'états

Théorème 3.8 *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 3.9 *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* $L(M) = L(M')$.

4 le lambda-calcul

Le λ-calcul a été inventé en 1930 par A. Church. Comme on va le voir, le λ-calcul est un outil qui permet d'exprimer des fonctions mathématiques. Les objets sur lesquels s'appliquent ces fonctions sont aussi des fonctions. Elles peuvent être, soient les fonctions elles-mêmes (on parlera alors du λ-calcul non-typé) ou sur des fonctions de type donné (on parlera alors de λ-calcul typé). Le λ-calcul a de nombreuses applications au niveau informatique, entre autre pour exprimer la sémantique des langages de programmation et pour la conception de preuves d'exactitudes de programmes.

Nous considérerons dans ce cours le λ-calcul d'un point de vue purement calculatoire, c'est-à-dire qu'on ne se posera pas vraiment la question de savoir si on s'en sert de façon typé ou non-typé. Ces deux notions ne change strictement rien à la méchanique du λ-calcul.

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:

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

La λ-expression $(E_1)E_2$ exprime l'application de la fonction E_1 à E_2.

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 apparaîssant immédiatement à droite du symbole "λ" dans E. Les λ-expressions closes n'ont donc que des variables liées.

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 par cette 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, en prenant soin de renommer chaque variable de E qui serait libre dans X. On évite ainsi de rendre liée une variable de X qui serait libre. On définit $E' = [X/x]E$, la substitution de x par X dans E par:

- si E est une variable alors $E' = \begin{cases} X & \text{si } E = x \\ E & \text{si } E \neq x \end{cases}$.
- si E est de la forme $(E_1)E_2$ alors $E' = ([X/x]E_1)[X/x]E_2$.
- 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 la première variable (en ordre lexicographique) qui ne soit ni dans X, ni dans Y.

Voyons quelque exemples:

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* $[\lambda x.(f)x/x]\lambda f.\lambda y.(f)(x)y$ *?*

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

α-conversion

Le choix du premier nom de variable dans l'ordre lexicographique qui ne soit liée ni dans X, ni dans Y comme variable lors de la substitution est totalement arbitraire. On aurait pu aussi bien choisir n'importe quel nom de variable qui ne soit lié ni dans X, ni dans Y. À cause de ce fait nous identifions ensemble toutes les fonctions du λ-calcul qui ne diffèrent que par un changement de variable qui 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. En général on ne porte pas attention aux noms de variables utilisées mais seulement aux fonctions définies par les λ-expressions. Le procédé de conversion par changement de variable est connu sous le nom d'α-conversion:

Définition 4.2 (α-conversion) *si y n'est pas libre dans $\lambda x.X$, alors*

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

En fait, l'α-conversion constitue une relation d'équivalence sur les λ-expressions.

β-conversion

La β-conversion permet d'évaluer un niveau d'une λ-expression. Quand on a une expression de la forme $E = (\lambda x.X)Y$, l'évaluation d'un niveau de E correspond à appliquer $\lambda x.X$ à Y. Ceci s'exprime par le procédé suivant:

Définition 4.3 (β-conversion)

$$(\lambda x.X)Y \mapsto_\beta [Y/x]X$$

Ce type de conversion permet de réduire une λ-expression pour éventuellement obtenir une forme la plus simple possible. Une λ-expression de la forme qui nous intéresse ($E = (\lambda x.X)Y$) est appelée **rédex**. La **réduction** d'une λ-expression consiste à poursuivre la β-conversion jusqu'à ce qu'aucun rédex n'appraisse dans E. On dit d'une λ-expression qui ne contient pas de rédex qu'elle est en **forme normale**.

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

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

laquelle est en forme normale.

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

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

laquelle est en forme normale.

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*-conversion 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* $(E)E$*:*

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

Il est vrai que $(E)E$ *n'a pas de forme normale non plus, mais ceci est beaucoup moins évident qu'à l'exemple précédent.*

η-conversion

Il existe aussi une autre forme de conversion, dite η-conversion qui permet de supprimer une variable inutile d'une λ-expression. Quand on a une expression de la forme $E = \lambda x.(Y)x$, sans que x n'apparaisse libre dans Y, il va de soit que λx et x peuvent être éliminées. Ceci s'exprime par le procédé suivant:

Définition 4.4 (η-conversion) *si* x *n'est pas libre dans* Y*, alors*

$$\lambda x.(Y)x \mapsto_\eta Y$$

Il est également possible de définir une notion similaire à la forme normale, mais dans le cadre de notre étude du λ-calcul nous n'aurons aucunement besoin de l'η-conversion. Nous ne nous avancerons donc pas plus loin dans cette direction.

Unicité de la forme normale

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

Soit $\mapsto$ une relation binaire. On dit que $\mapsto$ a la propriété de Church-Rosser, si

$$\forall E, E_1, E_2 \ [\text{si } E \mapsto E_1 \text{ et } E \mapsto E_2 \text{ alors } \exists N \ [E_1 \mapsto N \text{ et } E_2 \mapsto N]]$$

Soit $\mapsto_\beta^*$ la fermeture réflexive transitive de $\mapsto_\beta$, c'est-à-dire que $E_1 \mapsto_\beta^* E_2$ veut dire ou bien que $E_1 \mapsto_\alpha E_2$, ou bien que qu'il existe E_3 telque $E_1 \mapsto_\beta E_3$ et $E_3 \mapsto_\beta^* E_2$.

Théorème 4.5 (Church-Rosser) *la relation $\mapsto_\beta^*$ possède la propriété de Church-Rosser.*

On peut déduire de ce théorème l'important corollaire suivant:

Corollaire 4.6 *Soit E une λ-expression et soient N une λ-expression normale telle que $E \mapsto_\beta^* N$. Pour toute N' normale telle que $E \mapsto_\beta^* N'$on a que $N' \mapsto_\alpha N$.*

C'est-à-dire que N, la forme normale de E, est essentiellement unique. Par contre, le théorème ne dit pas que toute suite de β-conversions de E va finir par N car il est possible que certaines de ces suites 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

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

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

Le théorème affirme seulement que toute suite de conversions 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 telle suite existe. On définit la **réduction normale** d'une λ-expression comme celle qui applique toujours la β-conversion au rédex le plus à gauche.

Théorème 4.7 (Curry) *Soit E une λ-expression. S'il existe N en forme normale telle que $E \mapsto_\beta^* N$ alors le 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 traite que des fonctions. Il nous faut donc définir les entiers dans le λ-calcul sous forme de fonctions.

entiers et fonctions entières

Définition 4.8 *On définit les entiers comme suit:*

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

On appelle ces fonctions les **entiers de Church**. Ils nous serviront dorénavent à représenter les entiers. On peut maintenant définir des fonctions sur les entiers en utilisant ceux de Church. L'évaluation de ces fonctions va se faire au moyen de la β-conversion. Remarquez que tous les entiers de Church sont des λ-expressions en forme normales. Donc pour évaluer une fonction F sur donnée $\overline{n}$, on va procéder à la réduction normale de l'expression $(F)\overline{n}$. Il est garantie par le théorème de Curry que si la réponse du calcul est un entier de Church alors la réduction normale se terminera.

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:*

$$\begin{aligned}
(SUC)\overline{n} \quad &= \quad (\lambda x.\lambda y.\lambda z.(y)((x)y)z)\lambda x.\lambda y.\underbrace{(x)(x)(x)...(x)(x)(x)}_{n}y \\
&\mapsto_\beta \quad \lambda y.\lambda z.(y)((\lambda x.\lambda y.\underbrace{(x)(x)(x)...(x)(x)(x)}_{n}y)y)z
\end{aligned}$$

$$
\begin{aligned}
&\mapsto_\beta \quad \lambda y.\lambda z.(y)(\lambda a.\underbrace{(y)(y)(y)...(y)(y)(y)}_{n}a)z \\
&\mapsto_\beta \quad \lambda y.\lambda z.(y)\underbrace{(y)(y)(y)...(y)(y)(y)}_{n}z \\
&\mapsto_\beta \quad \lambda y.\lambda z.\underbrace{(y)(y)(y)...(y)(y)(y)}_{n+1}z \\
&\mapsto_\alpha \quad \overline{n+1}
\end{aligned}
$$

Il est également possible de définir de nombreuse fonctions à valeurs entières de $\bigcup_{k\in N}\{f : N^k \rightarrow N\}$ à l'aide du λ-calcul. Une fonction de $N^k \rightarrow N$ 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:*

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

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

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 4.9 *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 de simple 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-conversion de $(((IF)B)E_1)E_2$ va mener à E_1 si $B = VRAI$ et à E_2 si $B = FAUX$.

On va également pouvoir utiliser les booléens et 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*

$$SIG = \lambda n.(((IF)(ZERO?)n)\overline{0})\overline{1}$$

récursivité en λ-calcul

Considérons la λ-expression $MULT = \lambda x.\lambda y.\lambda z.(x)(y)z$ qui fait la multiplication de deux entiers de Church, et la λ-expression

$$PRED = \lambda n.(((n)\lambda p.\lambda u.((u)(SUC)(p)VRAI)(p)VRAI)\lambda u.((u)\overline{0})\overline{0})FAUX$$

qui calcule le prédécesseur d'un entier de Church et supposons que l'on veuille definir une λ-expression pour la factorielle $FACT$. On pourrait pour cela essayer de résoudre l'équation

$$FACT \mapsto_\beta^* \lambda n.(((IF)(ZERO?)n)\overline{1})((MULT)n)(FACT)(PRED)n$$

et trouver une λ-expression $FACT$ la satisfaisant. Posons d'abord

$$H = \lambda f.\lambda n.(((IF)(ZERO?)n)\overline{1})((MULT)n)(f)(PRED)n$$

ce que l'on voudrait obtenir est une expression pour $FACT$ qui satisfasse

$$FACT \mapsto_\beta (H)FACT.$$

En général, on dira que $FACT$ est un **point fixe** de H. Il se trouve qu'en λ-calcul il est très simple de trouver un point fixe pour une expression quelconque. Cela découle de l'existence d'un opérateur Y , dit de point fixe, qui a la propriété extraordinaire que pour tout E, il y a G tel que

$$(Y)E \mapsto_\beta G \mapsto_\beta (E)G.$$

Ce merveilleux Y est tout simplement

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

puisque

$$\begin{array}{rcll} (Y)E & = & (\lambda h.(\lambda x.(h)(x)x)\lambda x.(h)(x)x)E & \\ & \mapsto_\beta & (\lambda x.(E)(x)x)\lambda x.(E)(x)x & (= G) \\ & \mapsto_\beta & (E)(\lambda x.(E)(x)x)\lambda x.(E)(x)x & \\ & = & (E)G. & \end{array}$$

On pose donc finalement

$$(Y)H \mapsto_\beta^* FACT.$$

Cet exemple nous montre une technique générale permettant de définir récursivement des fonctions en λ-calcul.

5 théorèmes d'équivalence

Dans cette section nous allons étudier l'équivalence entre les notions de fonctions partielles récursives, Turing-semi-calculables et λ-représentables. Voyons donc d'abord cette dernière notion:

fonctions λ-représentables

On définit maintenant les fonctions du λ-calcul qui sont **λ-représentables**. Quand le contexte est clair on écrira $F\overline{n_1}\overline{n_2}...\overline{n_k}$ 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*

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

Théorème 5.2 *Les fonctions récursives partielles sont λ-représentables.*

Preuve: Notons pour commencer que les fonctions $Z(n)$, $S(n)$ et U_i^k sont λ-représentables car les λ-expressions suivantes leur sont équivalente:

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

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

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

On montre ensuite que les fonctions λ-représentables sont fermées sous l'action de la composition.

Lemme 5.3 *Si ψ_1, ψ_2,...,ψ_m dans $N^k \to N$ et χ dans $N^m \to N$ sont λ-représentables par des λ-expressions Ψ_1, Ψ_2,..., Ψ_m et Ξ alors la composition*

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

l'est aussi.

Preuve du lemme 5.3: Dans le cas où toutes les fonctions ψ_i et χ sont totales ce lemme est presqu'évident puisqu'il suffit de prendre l'expression

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

pour λ-représenter

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

Malheureusement, cette expression n'est plus un bon choix 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 *Par exemple si on choisit $\chi = U_1^2$, $\psi_1 = U_1^1$ et ψ_2 indéfinie partout. Dans ce cas, $\chi(\psi_1, \psi_2)$ n'est nulle part définie alors que $\lambda n.((\Xi)(\Psi_1)n)(\Psi_2)n$ est normalisable et se réduit à $\lambda n.n$, malgré que $(\Psi_2)n$ ne le soit pas.*

Ce qu'il nous faut est une λ-expression qui ne sera pas normalisable dès que l'une ou l'autre des $\Psi_i N_1...N_k$ n'est pas normalisable ou que $\Xi M_1...M_m$ ne l'est pas, où N_i est une λ-expression dont la forme normale est un entier de Church, et M_i la forme normale de $\Psi_i N_1...N_k$. Cet approche découle de la règle de composition des fonctions donnée à la section 2.1. Afin de construire une telle λ-expression nous aurons besoin d'un certain nombre de lemmes.

Étant donné deux λ-expression E_1 et E_2 on dénote $E_1 \circ E_2$ l'expression $\lambda x.(E_1)(E_2)x$. On note aussi $E_1 \rightsquigarrow_\beta E_2$ pour dire que E_2 s'obtient à partir de E_1 par β-conversion du rédex le plus à gauche.

Lemme 5.4 *Pour $k > 0$ on a:*

$$\underbrace{(\lambda g.g \circ s)(\lambda g.g \circ s)...(\lambda g.g \circ s)}_{k} h \rightsquigarrow_\beta^* \lambda x.(h) \underbrace{(s)(s)...(s)}_{k} x.$$

Ce lemme se montre par récurrence sur k.

Lemme 5.5 *Soient Φ et E deux λ-expressions. On pose $< \Phi, E >= (((E)\lambda g.g \circ SUC)\Phi)\overline{0}$. Si $\overline{n}$ est la forme normale de E, alors la forme normale de $< \Phi, E >$ est la même que celle de $(\Phi)\overline{n}$.*

Preuve du lemme: Puisque E a $\overline{n}$ pour forme normale on a

$$\begin{aligned}(E)\lambda g.g \circ SUC &\leadsto_\beta^* (\lambda x.\lambda y.\underbrace{(x)(x)...(x)}_{n} y)\lambda g.g \circ SUC \\ &\leadsto_\beta \lambda y.\underbrace{(\lambda g.g \circ SUC)(\lambda g.g \circ SUC)...(\lambda g.g \circ SUC)}_{n} y \\ &\leadsto_\beta^* \lambda y.\lambda x.(y)\underbrace{(SUC)(SUC)...(SUC)}_{n} x\end{aligned}$$

Si on applique maintenant cette expression aux paramètres Φ et $\overline{0}$ on obtient $(\Phi)\underbrace{(SUC)(SUC)...(SUC)}_{n}\overline{0}$ qui aura la même forme normale que $(\Phi)\overline{n}$. ■

Remarquons aussi que si E ou Φ n'ont pas de forme normale, alors $< \Phi, E >$ n'en aura pas non plus.

Définition 5.6 *On définit* $< \Phi, E_1, E_2, ..., E_m >$ *par*

$$< \Phi, E_1, E_2, ..., E_m > = << \Phi, E_1, E_2, ..., E_{m-1} >, E_m > .$$

Grâce au dernier lemme, nous pouvons maintenant donner une expression λ-représentant $\chi(\psi_1, \psi_2, ..., \psi_m)$. Considérons la λ-expression

$$\Gamma = \lambda n_1.\lambda n_2....\lambda n_k. < \Xi, \Psi_1 n_1...n_k, \Psi_2 n_1...n_k, ..., \Psi_m n_1...n_k > .$$

On sait par le dernier lemme que $(...((\Gamma)N_1)N_2)...)N_k$ aura la même forme normale que

$$(...(((\Xi)\Psi_1 N_1...N_k)\Psi_2 N_1...N_k)...)\Psi_m N_1...N_k$$

quand tous les $\Psi_i N_1...N_k$ sont normalisables et que $(...((\Xi)M_1)...)M_m$ l'est aussi, où N_i est une λ-expression dont la forme normale est un entier de Church, et M_i la forme normale de $\Psi_i N_1...N_k$. Aussi, si l'un d'eux n'est pas normalisable alors $(...((\Gamma)N_1)N_2)...)N_k$ ne sera pas normalisable non plus. ■

On montre maintenant que l'ensemble des fonctions λ-représentables est fermé par l'action de la récursion primitive.

Lemme 5.7 *Si ψ (totale) dans $N^{k+2} \to N$ et χ (totale) dans $N^k \to N$ sont λ-représentables par des λ-expressions Ψ et Ξ alors la fonction φ définie par récursion primitive à partir de χ et ψ est aussi λ-représentable.*

Preuve du lemme 5.7: Cherchons une λ-expression Φ pour λ-représenter φ. Considérons

$$\Phi \mapsto_\beta^* \lambda m.\lambda n_1....\lambda n_k.(((IF)(ZERO?)m)\Xi n_1 n_2...n_k)E_\Phi$$

avec

$$E_\Phi = (...(((((\Psi)(...((((\Phi)(PRED)m)n_1)n_2)...)n_k)(PRED)m)n_1)n_2)...)n_k.$$

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

$$H = \lambda f.\lambda m.\lambda n_1....\lambda n_k.(((IF)(ZERO?)m)\Xi n_1 n_2...n_k)E_f$$

et $(Y)H \mapsto_\beta^* \Phi$ pour obtenir un Φ satisfaisant. ■

Finalement, on montre qu'il en va de même avec la minimalisation.

Lemme 5.8 *Si χ (totale) dans $N^{k+1} \to N$ est λ-représentable par une λ-expression Ξ alors la fonction φ définie par minimalisation à partir de χ est aussi λ-représentable.*

Preuve du lemme 5.8: Cherchons une λ-expression Φ pour λ-représenter φ. Considérons d'abord

$$MIN \mapsto_\beta^* \lambda m.\lambda n_1....\lambda n_k.(((IF)(ZERO?)\Xi m n_1...n_k)m)F_{MIN}$$

avec

$$F_{MIN} = (...((((MIN)(SUC)m)n_1)n_2)...)n_k.$$

Cette λ-expression a la propriété que $(...(((MIN)\overline{0})\overline{n_1})...)\overline{n_k} \mapsto_\beta^* \overline{\varphi(n_1, n_2, ..., n_k)}$ quand cette valeur existe et n'a pas de forme normale quand elle n'existe pas. Ceci vient du fait que la réduction normale de $(...(((MIN)\overline{m})\overline{n_1})...)\overline{n_k}$ donne $(...(((MIN)\overline{m+1})\overline{n_1})...)\overline{n_k}$ quand $\chi(m, n_1, n_2, ..., n_k) \neq 0$ et donne m quand au contraire $\chi(m, n_1, n_2, ..., n_k) = 0$. On sait comment résoudre cette équation à l'aide de l'opérateur de point fixe Y; il suffit de poser

$$H = \lambda f.\lambda m.\lambda n_1....\lambda n_k.(((IF)(ZERO?)\Xi m n_1...n_k)m)F_f$$

et $(Y)H \mapsto_\beta^* MIN$ pour obtenir $\Phi = \lambda n_1.\lambda n_2....\lambda n_k.(...(((MIN)\overline{0})n_1)...)n_k$.
■

La preuve du théorème découle des trois lemmes 5.3, 5.7 et 5.8. ■

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 travaille de M contiendra donc $\lambda, a, b, c, ..., x, y, z, ', ., (,)$. Pour faire la conversion, M n'a uniquement 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 remplacent explicitement par E'. Pour éviter tout conflict 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-conversion 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 calcule f. Supposons que M compte e états et t symboles de travail. 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 DI initiale $q_0 0^{n_1}10^{n_2}1...10^{n_k}$. Pour ceci, il va écrire le nombre qui en base $t + e + 1$ correspondant à

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

(les nombres $1, 2, ..., t$ sont associés aux symboles de travail de M avec 1 qui représente 1 et 2 qui représente 0, et le nombres $t+1, t+2, ..., t+e$ représentent les états de M). 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).

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 DI en la DI suivante selon les actions de M. Quand M s'arrête ou si $a = 0$, $\psi(a, b, c) = 0$. Considérons maintenant la fonction suivante définie par minimalisation partielle:

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

Si M s'arrête sur donnée $0^{n_1}10^{n_2}1...10^{n_k}$ alors $\varphi(pred(\delta(\vec{n})), \vec{n})$ sera définie et aura pour valeur la dernière DI 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 avec (t') placé n'importe où dans le mot, pour $t' > t$ représentant l'état de M avant son arrêt. La fonction ν n'est pas définie pour les nombres n'ayant pas la bonne forme. 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).*

6 décidabilité et indécidabilité

On ne va maintenant s'intéresser qu'aux machines de Turing et aux ensembles qu'elles permettent de reconnaître. Puisque l'on sait que les différentes notions de fonctions calculable sont équivalentes, on peut maintenant redéfinir les notions d'ensembles récursifs et récursivement énumérables en termes de machines de Turing plutot que de fonctions récursives. Ceci nous permet de généraliser la définition:

Soit Σ un alphabet fini et $L \subseteq \Sigma^*$, un langage sur Σ. Soit λ la fonction caractéristique de L

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

et soit φ la fonction partielle caractéristique de L

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

Définition 6.1 *L est un <u>langage récursif</u> s'il existe une machine de Turing M qui s'arrête sur tous les mots de Σ^*, telle que $M(\omega)$ accepte si et seulement si $\lambda(\omega) = 1$.*

Définition 6.2 *L est un <u>langage récursivement énumérable</u> s'il existe une machine de Turing M telle que $M(\omega)$ accepte si et seulement si $\varphi(\omega) = 1$.*

On va tout particulièrement s'intéresser aux langages qui "encodent" les exemplaires des problèmes du type:"Quelles sont les machines de Turing qui acceptent le mot vide?". Cette question peut-être formulée en terme d'appartenance à un langage sur $\{0, 1\}$ en encodant les machines de Turing par des mots constitués de 0s et de 1s. Par exemple, le langage correspondant à la question précédente, est r.é. mais non récursif. C'est dire qu'il exsite un algorithme pour dire quelles machines de Turing acceptent le mot vide mais qu'il n'existe pas d'algorithme pouvant dire lesquelles ne l'acceptent pas.

6.1 propriétés des langages récursifs et r.é.

Pour les machines de Turing qui s'arrêtent sur tous les mots, on peut toujours la récrire avec deux états terminaux possibles nommés *oui* et *non*, pour dire que la machine accepte ou rejète le mot en donnée (*oui* est définit même si la machine ne s'arrête pas toujours). On va construire des preuves à 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 6.3 *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 6.4 *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 combiner séquentiellement comme précédemment, car M_1 peut ne jamais s'arrêter. Il faut donc combiner "parrallèlement" 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 rejète si M_1 et M_2 atteignent l'état *non*.
■

Théorème 6.5 *Si un langage L et son complément $\overline{L}$ sont tous les deux r.é. alors L est récursif.*

Preuve: Soit M_1 une machine qui accepte L et M_2 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 rejète si c'est M_2 qui accepte. M va nécessairement finir par s'arrêter que sa donnée soit ou non dans L.
■

6.2 codage des machines de Turing

On va maintenant se donner un codage des machines de Turing de façon à décrire les décrire sous forme de mot sur $\{0,1\}$. Par le théorème 3.3 l'on sait que tout langage sur $\{0,1\}$ 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 consière donc que les machines M de la forme

$$M = \{Q, \{0,1\}, \{0,1,B\}, \delta, q_1, B, \{q_2\}\}$$

avec $Q = \{q_1, q_2, ..., q_n\}$ un ensemble fini d'états. Tous les langages récursifs 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 simplement par sa fonction δ. Afin de simplifier notre codage, on identifie aussi $\{0,1,B\}$ avec $\{x_1, x_2, x_3\}$ et $\{\triangleleft, \triangleright, \bigtriangledown\}$ avec $\{d_1, d_2, d_3\}$.

Avec ces conventions, on va maintenant représenter chaque transition de la forme $\delta(q_i, x_j) = (q_k, x_l, d_m)$ par le mot $0^i10^j10^k10^l10^m$. Alors la description complète d'un machine pourra s'écrire

$$111code_1 11code_2 11...11code_r 111$$

où les $code_n, 1 \leq n \leq r$, correspondent aux différentes transitions données par δ et codées par la règle précédente. L'ordre dans laquelle les différentes transition données par δ sont présentées dans le codage n'a pas d'importance.

On dénote par $< M >$ un tel codage de M. Il existe plusieurs codages corrects de M et tous sont équivalents. Par contre, chaque codage définit uniquement une machine. Si plusieurs transitions existent pour le même couple (q_i, x_j) alors la machine représentée sera non-déterministe. Dans ce qui suit, on va s'intéresser seulement aux codages des machines déterministes.

Exemple 6.1 *Soit* $M = \{\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_2\}\}$ *avec* δ *donnée par:*

$$\begin{array}{rcl} \delta(q_1, 1) & = & (q_3, 0, \triangleright) \\ \delta(q_3, 0) & = & (q_1, 1, \triangleright) \\ \delta(q_3, 1) & = & (q_2, 0, \triangleright) \\ \delta(q_3, B) & = & (q_3, 1, \triangleleft). \end{array}$$

Un codage de M serait:

$<M>$=11101001000101001100010101001001100010010010100110001000100010010111

De la même façon, on dénotera $< M, \omega >$ la concaténation de $< M >$ et ω. Ceci nous permettra de dénoter les paires de machines et de leurs données. Un mot de la forme $< M, \omega >$ peut-être décodé de façon unique, grace à la forme spéciale du codage de M. Le mot correspondant à $< M, \omega >$ devrait commencer par 111 puis avoir un certain nombre de codage de transitions, puis 111 une seconde fois avant ω qui commence immédiatement après. Par exemple,

111010010001010011000101010010011000100100101001100010001000100101111101

serait un codage correct de $< M, 1101 >$ pour la machine M de l'exemple 6.1.

6.3 un langage non r.é.

Supposons que l'on énumèrent tous les mots sur $\{0, 1\}$ dans l'ordre canonique (en commençant par les plus courts, puis en ordre numérique pour chaque longueur), et que l'on nomme ω_i le $i^{\text{ième}}$ mot dans l'énumération. Supposons maintenant que l'on énumèrent les machines, en nommant M_j la machine décrite par l'entier j interpété 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 à la figure 12 une protion de cette table hypothétique.

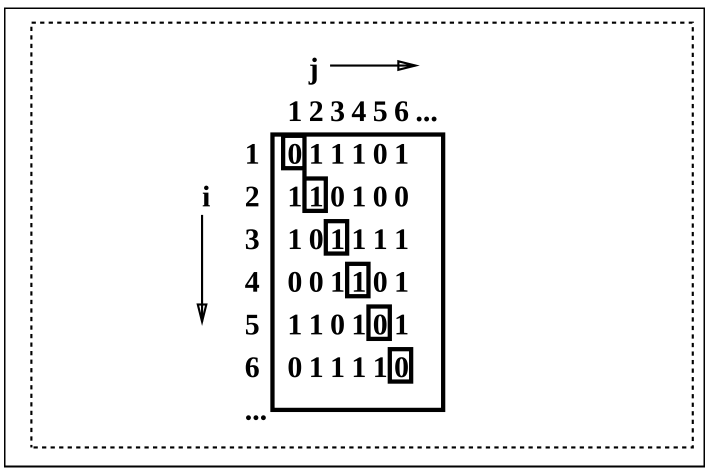

Figure 12: table hypothétique de $\omega_i \in L(M_j)$

On construit maintenant le langage L_d 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 6.6 *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 $L(M) = L_d$.
$\blacksquare$

6.4 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 6.7 *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. La tête de lecture sur ce ruban sera utilisé pour trouver les transitions de la forme (q_i, x_j) dans la description de M. Le second ruban de $\mathcal{M}$ correspondra au ruban de M (on y copira ω au début). Enfin, le troisième ruban sera utilisé pour mémoriser les états de M. Le comportement de $\mathcal{M}$ sera:

1. Vérifie le format de la donnée sur le ruban 1, que chaque couple (q_i, x_j) n'apparaît qu'une fois dans M et que chaque code $0^i10^j10^k10^l10^m$ est correctement constitué (c'est-à-dire que $1 \leq j, l, m \leq 3$).

2. Copie ω sur le ruban 2. Initialise le ruban 3 à 0 pour indiquer que M est dans l'état q_1, puis repositionne toutes les têtes en début de ruban.

3. Si le ruban 3 contient 00, signifiant que M est dans l'état q_2, alors $\mathcal{M}$ s'arrête en acceptant.

4. Supposons que le symbole que voit présentement $\mathcal{M}$ sur le ruban 2 soit x_j et que 0^i soit le mot courant sur le ruban 3. Une recherche sur le ruban 1 du sous-mot 110^i10^j mènera soit à une transition de la forme $0^i10^j10^k10^l10^m$, soit à aucune transition de ce type. Dans le premier

cas, $\mathcal{M}$ va remplacer x_j par x_l sur le ruban 2, remplacera le mot 0^i par 0^k sur le ruban 3 et se déplacera sur le ruban 2 dans la direction donnée par 0^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 donnée ω. Enfin, si M rejète ω en s'arrêtant, $\mathcal{M}$ en fera autant.
■

La machine $\mathcal{M}$ décrite ci-haut 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 3.3 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 6.2 faisant le même travail que $\mathcal{M}$. Appelons cette machine M_u. M_u est *la machine de Turing universelle.*

6.5 réductions

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 existe une machine M telle que $L = L(M)$ alors il existe une machine M' telle que $L' = L(M')$. On déduit par contraposition que M ne peut exister puisque M' n'existe pas.

Théorème 6.8 *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. Pour montrer que $\overline{L_u}$ n'est pas r.é. on montre que

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

Supposons qu'il existe une machine M pouvant reconnaître $\overline{L_u}$. On construit une machine M' qui accèpte L_d. Sur donnée ω, M' trouve un i tel que $\omega = \omega_i$. Avec ce i, M' construit la donnée $< M_i, \omega_i >$ et la donne à M. Si M accèpte $< M_i, \omega_i >$ alors M' accèpte ω sinon elle rejète. Il s'en suit que M' accèpte ω_i si et seulement si M_i n'accèpte pas ω_i, donc que $\omega_i \in L_d$. Mais, $\omega_i = \omega$,

donc M' accèpte exactement les mots dans L_d. Puisque M' ne peut exister (par le théorème 6.6), M ne peut exister non-plus.
■

le langage vide

Ce dernier théorème nous mène à la conclusion que L_u est indécidable. On va maintenant utiliser ce fait pour démontrer l'indécidabilité de plusieurs autres problèmes similaires.

Théorème 6.9 *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'accèpte. Considérons une machine non-déterministe M' qui sur donnée $< M >$ va choisir de façon non-déterministe un mot ω tel que $\omega \in L(M)$. Un tel mot existe puisque $L(M) \neq \emptyset$. Ensuite M' simule le travail de M sur donnée ω. M' accèpte si M accèpte. Bien sur, puisque ω est accèpté par M, M' va accèpter $< M >$. Remarquez aussi que s'il n'existe pas d'ω dans $L(M)$ alors M' n'acceptera pas $< M >$. Par l'existance de M', $\overline{L_\emptyset}$ est r.é..

On voudrait maintenant montrer que $L_\emptyset$ n'est pas r.é.. Supposons qu'il se soit, il exsite alors M' qui accèpte tous les $< M >$ telles que $L(M) = \emptyset$. On va utiliser ce fait pour obtenir une machine M'' pour $\overline{L_u}$. Sur donnée $< M, \omega >$, M'' construit une machine $M_{<M,\omega>}$ qui commence par ignorer sa donnée puis écrit ω sur son ruban, puis exécute M sur donnée ω. $M_{<M,\omega>}$ accèpte si et seulement si M accepte. Remarquez que $L(M_{<M,\omega>}) = \emptyset$ si $< M, \omega > \in \overline{L_u}$ et $L(M_{<M,\omega>}) = \Sigma^*$ si $< M, \omega > \in L_u$. Afin de décider si $< M, \omega > \in \overline{L_u}$, M'' exécute M' sur donnée $< M_{<M,\omega>} >$ et accepte si et seulement si M' accepte. Si M' sais reconnaître les machines exprimant le langage vide, alors elle saura dire quand $L(M_{<M,\omega>})$ est vide, donc quand $< M, \omega > \in \overline{L_u}$. Comme $\overline{L_u}$ n'est pas r.é., alors $L_\emptyset$ ne l'est pas non plus.
■

6.6 théorèmes de Rice

On considèr 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 *triviale* 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 décidable si $L_{\mathcal{L}}$ est récursif et indécibadle sinon.

Le théorème de Rice nous donne un critère très général mais très simple permettant de décider si une propriété des langages r.é. est décidable.

Théorème 6.10 (Rice) *Toute propriété non triviale $\mathcal{L}$ des langages r.é. sur $\{0,1\}$ est indécidable.*

Preuve: Soit $\mathcal{L}$ une propriété non triviale des langages r.é. sur $\{0,1\}$. Sans perdre de généralité $\emptyset \in \mathcal{L}$, sinon on considère $\{\text{langages r.é. sur } \{0,1\}\} - \mathcal{L}$. D'autre part, puisque $\mathcal{L}$ est non triviale, il existe un langage $L \in \{\text{langages r.é. sur } \{0,1\}\} - \mathcal{L}$ et une machine M_L qui reconnaît L. En principe si $\mathcal{L}$ était décidable, il existerait une machine $M_{\mathcal{L}}$ qui pourrait décider si $< M > \in L_{\mathcal{L}}$, pour toute machine M. En particulier, $M_{\mathcal{L}}$ devrait savoir reconnaître les machines M telles que $L(M) = \emptyset \in \mathcal{L}$ des machines M telles que $L(M) = L \in \{\text{langages r.é. sur } \{0,1\}\} - \mathcal{L}$. On va construire une machine $M_{\emptyset}$ qui sait reconnaître les machines $< M >$ telles que $L(M) = \emptyset$ des autres en forçant les machines qui acceptent autre chose à accepter exactement L. Voici comment procède $M_{\emptyset}$:

1. Sur donnée $< M >$, construire une machine M' qui commence par trouver de façon non-déterministe un mot ω tel que $\omega \in L(M)$ et qui vérifie ensuite ce fait. Une fois ceci complété, M' exécute M_L sur sa donnée ω'.

2. Si $M_{\mathcal{L}}$ accepte $< M' >$ alors accepter $< M >$, sinon rejeter.

Remarquez que M' a été construite de façon à ce que si $L(M) = \emptyset$ alors $L(M') = \emptyset$ car M' va continuellement chercher un mot $\omega \in L(M)$ lequel n'existe pas. D'autre part si $L(M) \neq \emptyset$ alors $L(M') = L$ car une fois $\omega \in L(M)$ trouvé, M' fait exactement comme M_L. Or, la machine $M_{\mathcal{L}}$ sait reconnaître les codes des machines pour $\emptyset$ des codes des machines pour L. Donc la machine $M_{\emptyset}$ va savoir reconnaître les machines pour $\emptyset$ des autres.

Puisque $M_\emptyset$ n'exsite pas, alors $M_\mathcal{L}$ n'existe pas non plus.
■

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

Exemple 6.2 *Les propriétés suivantes 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:

Théorème 6.11 (Rice) *Soit une propriété non triviale $\mathcal{L}$ 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 génère sur son ruban de travail le mot (possiblement infini)*

$$code_1\#code_2\#...$$

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

Idée de preuve: On montre d'une part que si 1), 2) ou 3) n'est pas satisfait alors $\overline{L_u}$ est r.é., par une réduction. D'autre part, on donne une façon explicite d'accepter les codes de machines $< M >$ tels que $L(M) \in \mathcal{L}$ en se basant sur le fait que $\mathcal{L}$ possède les propriétés 1), 2) et 3).

Corollaire 6.12 *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 respecté, sauf le second où la condition 2) n'est pas respectée et le dernier où la condition 3) n'est pas respectée.
■

Corollaire 6.13 *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.*

6.7 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 6.14 *Soit G la grammaire génératrice d'un langage algébrique L. Il est indécidable de savoir si G est ambigüe.*

Théorème 6.15 *Soit G la grammaire génératrice d'un langage algébrique L. Il est indécidable de savoir si $L = \Sigma^*$.*

Théorème 6.16 *Soient G_1, G_2 deux grammaires génératrices de langages algébriques L_1, L_2. Il est indécidable de savoir si $L_1 = L_2$.*

Théorème 6.17 *Soient G_1, G_2 deux grammaires génératrices de langages algébriques L_1, L_2. Il est indécidable de savoir si $L_1 \cap L_2 = \emptyset$.*

6.8 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 entends par là.

machines de Turing avec Oracle

On consière 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 machine M^A a le droite de passer dans l'état $q_?$ afin de savoir si le mot situé à droite de sa tête de lecture appartient ou non à un certain langage A. La machine se retrouve alors dans l'état q_{oui} si elle a fournie un mot dans A et se retrouve dans l'état q_{non} autrement. Elle continue ensuite son calcul. On dénote par $L(M^A)$ le langage accepté par M par rapport à A.

Remarquons que si A est un langage récursif, la machine M pourrait bien faire le travail sans utiliser A puisqu'il existe une machine décidant l'appartenance à A. Par contre, si A n'est pas r.é. 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 telle que $L(M^A) = L$. Un langage L est récursif par rapport à A s'il existe une machine M qui s'arrête toujours telle que $L(M^A) = L$. Deux langages oracles sont équivalents si ils sont mutuellement récursif l'un par rapport à l'autre.

hiérarchie

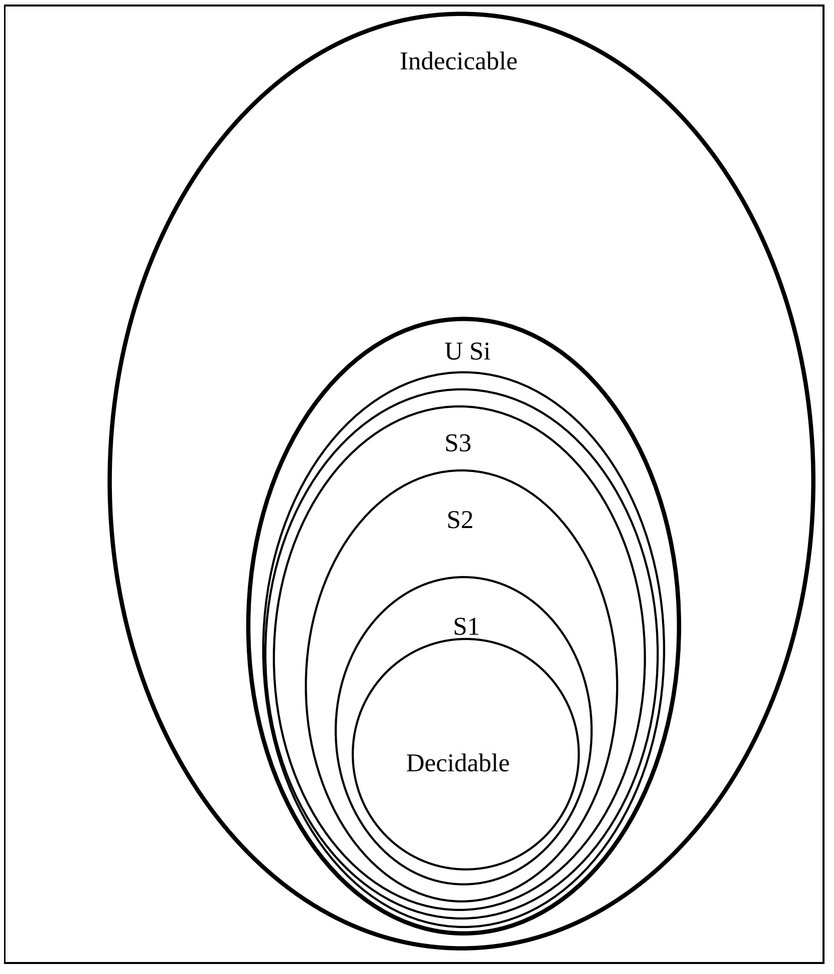

Figure 13: Hiérarchie de langages indécidables

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 langage et seulement un nombre dénombrable de machines. Considérons donc le langage

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

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

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

On va pouvoir maintenant dire "combien" indécidables certains problèmes sont en montrant qu'ils sont équivalent 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 l'un des S_i ne constituent qu'une infime partie de tous les langages possibles)

Exemple 6.3 *L_u est équivalent à S_1.*

Exemple 6.4 *$L_{\Sigma^*} = \{< M > | L(M) = \Sigma^*\}$ est équivalent à S_2.*

Exemple 6.5 *$L_{reg} = \{< M > | L(M)$ est régulier $\}$ est équivalent à S_3.*

7 mesures et classes de complexité

Nous étudions maintenant la *complexité* des langages sur $\{0, 1\}^*$. On va considérer principalement deux mesures de complexité, l'une liée á l'espace et l'autre liée au temps nécessaires à toute procédure de décision pouvant identifier les membres de ce langage. Il exsite 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 de regarder les deux plus importantes.

7.1 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.

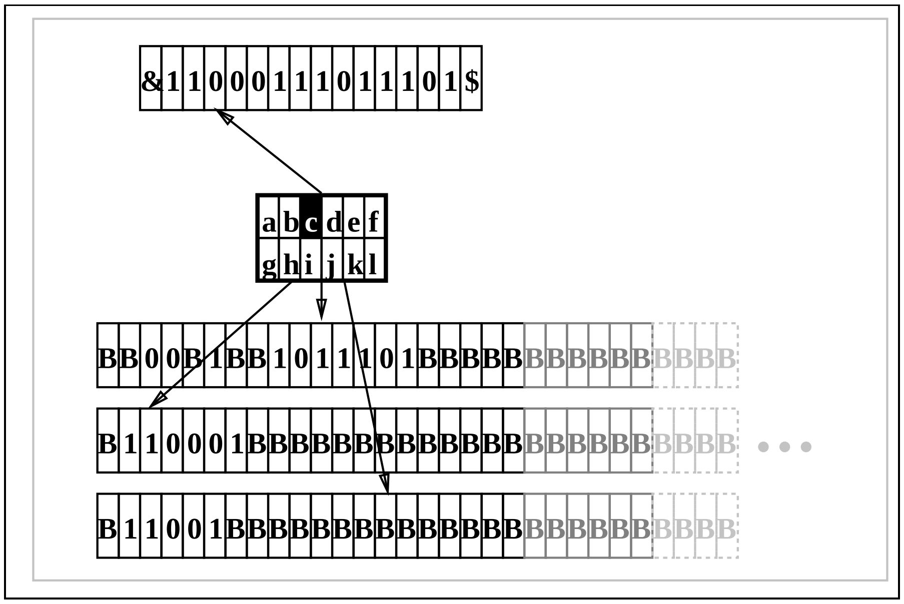

Figure 14: machine de Turing "off-line"

complexité d'espace

Si pour tout mot en donnée de longueur n, la machine M regarde un maximum de $S(n)$ cases sur chacun de ses rubans de travail on dira que M est une machine de *complexité d'espace* $S(n)$. On dit également d'un langage L qu'il est de complexité d'espace $S(n)$ s'il existe une machine M de complexité d'espace $S(n)$ telle que $L(M) = L$.

La raison pour laquelle on considère la complexité d'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 choisit, il nous sera possible de parler de langages de complexité d'espace $\log(n)$ par exemple.

complexité de temps

Considérons la machine de Turing M de la figure 15. M est une machine dite "bidirectionelle". Elle est équipée de k rubans de travail bidirectionels,

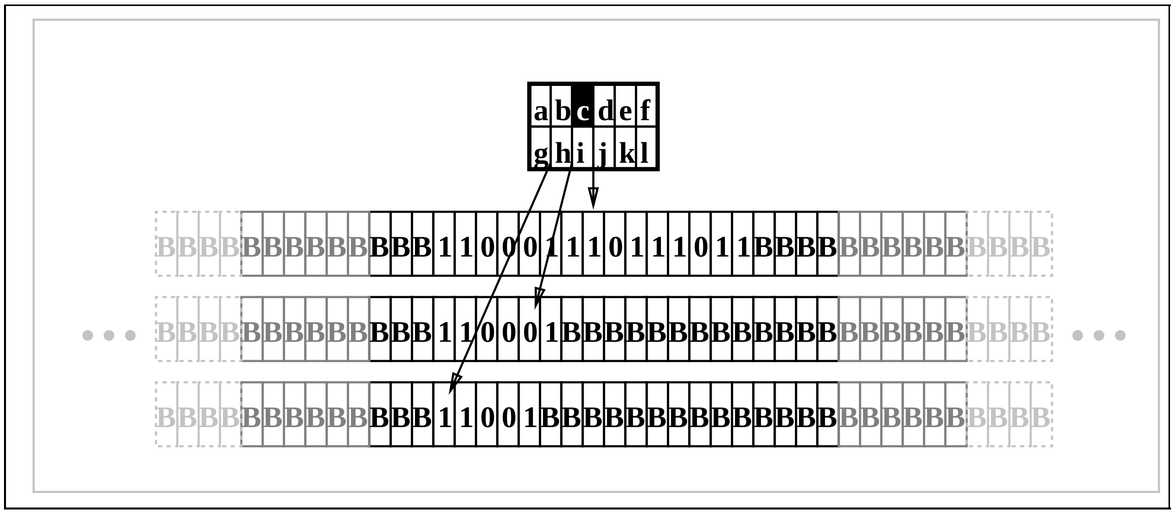

Figure 15: machine de Turing à rubans infinis bidirectionels

sa donnée étant sur son premier ruban.

Si pour tout mot en donnée de longueur n, la machine M fait un maximum de $T(n)$ transitions avant de s'arrêter on dira que M est une machine de *complexité de temps* $T(n)$. On dit également d'un langage L qu'il est de complexité de temps $T(n)$ s'il existe une machine M de complexité de temps $T(n)$ telle que $L(M) = L$.

Le choix de deux différents modèles pour la complexité d'espace et de temps permettra de faciliter l'expression de certaines preuves. Certaines variations auraient été possible sur le modèle mais pas nécessairement toute. Par exemple, nous pourrions pour $S(n) \geq n$ utiliser une machine standard avec un seul ruban en lecture/écriture sans pour autant changer l'ensemble des langages reconnus en espace $S(n)$. Néanmoins pour la complexité de temps, il existe une plus grande sensibilité au modèle.

non-déterminisme

Les mesures de complexité pour les machines non-déterministes considèrent les pires choix non-déterministes possibles. C'est-à-dire qu'une machine non-déterministe est de complexité d'espace (ou de temps) $S(n)$ (ou $T(n)$) si pour toutes les données de taille n et tous les choix non-déterministes possibles elle utilise au plus $S(n)$ cases de ses rubans (prend au plus $T(n)$ étapes) avant de s'arrêter.

7.2 classes de complexité

On dénote la famille des langages acceptés par des machines de complexité d'espace $S(n)$ par $\text{DESPACE}(S(n))$ et par $\text{NESPACE}(S(n))$ si les machines sont non-déterministes. De même, on dénote la famille des langages acceptés par des machines de complexité de temps $T(n)$ par $\text{DTEMPS}(T(n))$ et par $\text{NTEMPS}(T(n))$ si les machines sont non-déterministes.

7.3 accélération, compression, réduction

compression de rubans

Théorème 7.1 *Si L est accepté par une machine bornée en espace par $S(n)$ avec k rubans de travail, alors pour tout $c > 0$, L est accepté par une machine bornée en espace par $cS(n)$. Dit autrement,*

$$\forall c > 0 \ \ [\text{DESPACE}(S(n)) = \text{DESPACE}(cS(n))] \, .$$

Théorème 7.2

$$\forall c > 0 \ \ [\text{NESPACE}(S(n)) = \text{NESPACE}(cS(n))] \, .$$

Idée de preuves: On démontre que s'il existe une machine M_1 qui accèpte un langage L en espace $S(n)$, alors il existe une machine M_2 qui fait le même travail en prenant moins d'espace en compressant $1/c$ cases de M_1 en une case de M_2 (M_2 aura donc un alphabet beaucoup plus riche que celui de M_1).

accélération linéaire

Théorème 7.3 *Si L est accepté par une machine bornée en temps par $T(n)$ avec k rubans de travail, alors pour tout $c > 0$, L est accepté par une machine bornée en temps par $cT(n)$, à condition que $k > 1$ et que*

$$\liminf_{n\to\infty} \frac{T(n)}{n} = \infty.$$

Dit autrement,

$$\liminf_{n\to\infty} \frac{T(n)}{n} = \infty \Rightarrow \forall c > 0 \ \left[\textsc{Dtemps}(T(n)) = \textsc{Dtemps}(cT(n))\right].$$

Théorème 7.4

$$\liminf_{n\to\infty} \frac{T(n)}{n} = \infty \Rightarrow \forall c > 0 \ \left[\textsc{Ntemps}(T(n)) = \textsc{Ntemps}(cT(n))\right].$$

Théorème 7.5 *Si L est accepté par une machine bornée en temps par cn pour un $c > 1$ avec k rubans de travail, alors pour tout $\epsilon > 0$, L est accepté par une machine bornée en temps par $(1+\epsilon)n$, à condition que $k > 1$. Autrement dit,*

$$T(n) = cn > n \Rightarrow \forall \epsilon > 0 \ \left[\textsc{Dtemps}(T(n)) = \textsc{Dtemps}((1+\epsilon)n)\right].$$

Théorème 7.6

$$T(n) = cn > n \Rightarrow \forall \epsilon > 0 \ \left[\textsc{Ntemps}(T(n)) = \textsc{Ntemps}((1+\epsilon)n)\right].$$

Idée de preuves: On démontre que s'il existe une machine M_1 qui accèpte un langage L en temps $T(n)$, alors il existe une machine M_2 qui fait le même travail en prenant moins de temps en effectuant m (pour un m arbitraire mais fixé) étapes de M_1 en seulement 8 étapes de M_2.

réduction du nombre de rubans

Théorème 7.7 *Si L est accepté par une machine bornée en espace par $S(n)$ avec k rubans de travail, alors L est accepté par une machine bornée en espace par $S(n)$ n'utilisant qu'un seul ruban de travail.*

Théorème 7.8 *Si L est accepté par une machine bornée en temps par $T(n)$ avec k rubans de travail, alors L est accepté par une machine bornée en temps par $T^2(n)$ n'utilisant qu'un seul ruban de travail.*

Idée de preuves: 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 directement ces deux résultats.

Théorème 7.9 *Si L est accepté par une machine bornée en temps par $T(n)$ avec k rubans de travail, alors L est accepté par une machine bornée en temps par $T(n)\log T(n)$ n'utilisant que deux rubans de travail.*

Idée de preuves: Pour arriver à ce résultat on démontre comment simuler tous les rubans d'une machine M_1 sur un seul ruban d'une nouvelle machine M_2 en utilisant son second ruban comme mémoire temporaire. En faisant ce travail de façon judicieuse, on arrive a simuler les $T(n)$ étapes de M_1 en $T(n)\log T(n)$ étapes de M_2. Les mouvements des têtes de lectures de M_1 étant mimiqués par M_2 en décalant le contenu des différentes pistes de son premier ruban.

Théorème 7.10 *Les trois théorèmes précédents sont aussi valides pour les machines non-deterministes.*

7.4 théorèmes de hiérarchies

Théorème 7.11 *Étant donnée une fonction récursive $f(n)$, il existe toujours des langages L, L' tels que $L \notin \textsc{Dtemps}(f(n))$ et $L' \notin \textsc{Despace}(f(n))$.*

Idée de preuves: On démontre ceci par une diagonalization similaire à celle de la section 6.3.

Théorème 7.12 (Borodin) *Étant donné n'importe quelle fonction récursive $g(n) \geq n$, il existe une fonction récursive $S(n)$ telle que*

$$\textsc{Despace}(S(n)) = \textsc{Despace}(g(S(n)))$$

Théorème 7.13 (Blum) *Étant donné n'importe quelle fonction récursive $g(n)$, il existe un langage récursif L telle que toute machine M_i accèptant L en espace $S_i(n)$ (ou en temps $T_i(n)$) peut être remplacée par une autre machine M_j accèptant aussi L mais en espace $S_j(n)$ (ou en temps $T_j(n)$) tel que*

$$g(S_j(n)) \leq S_i(n) \text{ presque partout}$$

$$(g(T_j(n)) \leq T_i(n) \text{ presque partout})$$

hiérarchie sur l'espace

Théorème 7.14 *Si S_2 est totalement constructible par rapport à l'espace, que $S_1(n), S_2(n) \geq \log_2 n$ et que*

$$\liminf_{n\to\infty} \frac{S_1(n)}{S_2(n)} = 0$$

alors $\textsc{Despace}(S_2(n)) \not\subseteq \textsc{Despace}(S_1(n))$.

Idée de preuves: On démontre ceci par une diagonalization similaire à celle de la section 6.3.

hiérarchie sur le temps

Théorème 7.15 *Si T_2 est totalement constructible par rapport au temps et que*

$$\liminf_{n\to\infty} \frac{T_1(n)\log T_1(n)}{T_2(n)} = 0$$

alors

$$\textsc{Dtemps}(T_2(n)) \not\subseteq \textsc{Dtemps}(T_1(n)).$$

Idée de preuves: On démontre ceci par une diagonalization similaire à celle de la section 6.3.

7.5 relations entre les mesures de complexité

Théorème 7.16 *On a que*

1. *Si $L \in \textsc{Dtemps}(f(n))$ alors $L \in \textsc{Despace}(f(n))$.*
2. *Si $L \in \textsc{Despace}(f(n))$ et que $f(n) \geq \log_2 n$ alors il existe un $c_L > 0$ tel que $L \in \textsc{Dtemps}(c_L^{f(n)})$.*
3. *Si $L \in \textsc{Ntemps}(f(n))$ alors il existe un $c_L > 0$ tel que $L \in \textsc{Dtemps}(c_L^{f(n)})$.*

Théorème 7.17 (Savitch) *Si $L \in \textsc{Nespace}(S(n))$ alors $L \in \textsc{Despace}(S^2(n))$ à condition que $S(n)$ soit totalement constructible par rapport à l'espace et que $S_2(n) \geq \log_2 n$.*

lemme de translation

Lemme 7.18 *Soit $S_1(n), S_2(n), f(n)$ des fonctions totalement constructibles par rapport à l'espace telles que $S_2(n), f(n) \geq n$. On a que*

$$\text{DESPACE}(S_1(n)) \subseteq \text{DESPACE}(S_2(n))$$

$$\Rightarrow \text{DESPACE}(S_1(f(n))) \subseteq \text{DESPACE}(S_2(f(n)))$$

et que

$$\text{NESPACE}(S_1(n)) \subseteq \text{NESPACE}(S_2(n))$$

$$\Rightarrow \text{NESPACE}(S_1(f(n))) \subseteq \text{NESPACE}(S_2(f(n))).$$

Lemme 7.19 *Soit $T_1(n), T_2(n), f(n)$ des fonctions totalement constructibles par rapport au temps telles que $T_2(n), f(n) \geq n$. On a que*

$$\text{DTEMPS}(T_1(n)) \subseteq \text{DTEMPS}(T_2(n))$$

$$\Rightarrow \text{DTEMPS}(T_1(f(n))) \subseteq \text{DTEMPS}(T_2(f(n)))$$

et que

$$\text{NTEMPS}(T_1(n)) \subseteq \text{NTEMPS}(T_2(n))$$

$$\Rightarrow \text{NTEMPS}(T_1(f(n))) \subseteq \text{NTEMPS}(T_2(f(n)))$$

8 réductions polynomiales et $\mathcal{N}P$-complétude

Nous abordons maintenant l'étude des classes de complexité jugées comme les plus importantes d'un point de vu théorique: $\mathcal{P}$ et $\mathcal{N}P$. Ces deux classes correspondent intuitivement aux langages pour lesquels on peut décider efficacement de l'appartenance ($\mathcal{P}$) ou simplement de vérifier l'appartenance étant donné une preuve ($\mathcal{N}P$).

8.1 les classes $\mathcal{P}$ et $\mathcal{N}P$

Définition 8.1

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

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

Remarquez que ces deux classes sont stables aux changement de modèle de machine de Turing (en excluant le non-déterminisme, bien sur). Le nombre de ruban de la machine utilisée ne peut faire qu'un changement quadratique dans le temps nécessaire pour accépter un langage par le théorème 7.8.

Une question de toute importance en informatique théorique moderne est à savoir si

$$\mathcal{P} \stackrel{?}{=} \mathcal{N}P.$$

Cette question exprime mathématiquement l'intuition selon laquelle il est plus difficile de trouver la réponse à une question que de la vérifier. Une réponse négative à cette question ($\mathcal{P} \neq \mathcal{N}P$) aurait des implications pratiques importante car de nombreux problèmes naturels ne peuvent être résolus efficacement que si $\mathcal{P} = \mathcal{N}P$ (ceci nous mènera à la notion de $\mathcal{N}P$-complétude).

8.2 la classe $\mathcal{P}$ESPACE

On peut définir similairement de telles classes par rapport à la complexité en espace:

Définition 8.2

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

$$\mathcal{N}P\textsc{espace} = \bigcup_{i \in N} \textsc{Nespace}(n^i)$$

Mais cette fois, pas de grande question ouverte car par le théorème de Savitch (7.17), il découle que

$$\mathcal{P}\textsc{espace} = \mathcal{N}P\textsc{espace}.$$

8.3 Relation sur ces classes

On a les relations suivantes avec certitude sur ces classes:

$$\mathcal{P} \subseteq \mathcal{N}P \subseteq \mathcal{P}\textsc{espace}$$

mais on ne sais pas si ces inclusions sont strictes ou non.

8.4 Réductions polynômiales

On introduit maintenant la notion de réduction polynômiale entre des langages récursifs.

Définition 8.3 *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*
2. *pour tout $x \in \Sigma_1^*$ on a que*

$$x \in L_1 \Leftrightarrow f(x) \in L_2$$

Théorème 8.4 *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 $f(x) \in L_2$, $|f(x)| \leq p(|x|)$) qui est borné par un autre polynôme $q(|x|)$.

8.5 $\mathcal{N}P$-complétude

Définition 8.5 *Un langage L est dit* $\mathcal{N}P$-complet *si*

1. $L \in \mathcal{N}P$
2. *pour tout langage* $L' \in \mathcal{N}P$, $L' \leq_{\mathcal{P}} L$

8.5.1 $\mathcal{N}P$-complétude de SAT

Exemple 8.1 *Soient $u_1, u_2, ..., u_m$ des noms de variables booléennes. 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 $\{x_1, x_2, ..., x_k\}$. Une clause C est* satisfaite *si au moins un des littéraux de C est vrai. Une collection de clauses $\mathcal{C} = \{c_1, c_2, ..., c_n\}$ est* satisfaisable *s'il existe une assignation des variables $u_1, u_2, ..., u_m$ qui satisfasse chaque clause de $\mathcal{C}$.*

Théorème 8.6 (Cook) *Le problème suivant, nommé SAT, est $\mathcal{N}P$-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?*

Preuve:

1. *$SAT \in \mathcal{N}P$: il suffit de choisir la valeur des variables et de vérifier que toutes les clauses sont satisfaites avec ce choix de variables.*
2. *Pour tout $L \in \mathcal{N}P$, $L \leq_{\mathcal{P}} SAT$: Puisque $L \in \mathcal{N}P$, il existe une machine 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 décrivant le comportement de M pendant un temps $p(|x|)$ qui sera satisfaisable si et seulement si il existe des choix non-déterministes faisant accèpter x par M.*

■

La plus grande difficulté avec la $\mathcal{N}P$-complétude fut de démontrer l'existence d'un premier langage $\mathcal{N}P$-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 8.7 *Un langage L est $\mathcal{N}P$-*complet *si*

1. $L \in \mathcal{N}P$
2. *il existe un langage $\mathcal{N}P$-*complet *L' tel que $L' \leq_{\mathcal{P}} L$*

8.5.2 $\mathcal{N}P$-complétude de $3SAT$

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. On montre donc

Théorème 8.8 *Le problème $3SAT$ est $\mathcal{N}P$-complet.*

Preuve:

1. $3SAT \in \mathcal{N}P$: il suffit de choisir la valeur des variables et de vérifier que toutes les clauses sont satisfaites avec ce choix de variables.
2. $SAT \leq_{\mathcal{P}} 3SAT$:
   Soient U et $\mathcal{C}$, les données d'un problème de SAT. On décrit U' et $\mathcal{C}'$ constituant un problème de $3SAT$ équivalent. Posons, $U = \{u_1, u_2, ..., u_m\}$ et $\mathcal{C} = \{c_1, c_2, ..., c_n\}$. Pour chaque clause c_i on aura:
   - $\mathcal{C}_i = \{\{x_i, y_i^1, y_i^2\}, \{x_i, \overline{y_i^1}, y_i^2\}, \{x_i, y_i^1, \overline{y_i^2}\}, \{x_i, \overline{y_i^1}, \overline{y_i^2}\}\}$ et $U_i = \{y_i^1, y_i^2\}$ si $c_i = \{x_i\}$
   - $\mathcal{C}_i = \{\{x_i^1, x_i^2, y_i\}, \{x_i^1, x_i^2, \overline{y_i}\}\}$ et $U_i = \{y_i\}$ si $c_i = \{x_i^1, x_i^2\}$
   - $\mathcal{C}_i = c_i$ et $U_i = \emptyset$ si $c_i = \{x_i^1, x_i^2, x_i^3\}$
   - $\mathcal{C}_i = \{\{x_i^1, x_i^2, y_i^1\}, \{\overline{y_i^1}, x_i^3, y_i^2\}, ..., \{\overline{y_i^{k-4}}, x_i^{k-2}, y_i^{k-3}\}, \{\overline{y_i^{k-3}}, x_i^{k-1}, x_i^k\}\}$ 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 $k > 3$

On pose finalement,

$$U' = U \bigcup \bigcup_{i=1}^{n} U_i \text{ et } \mathcal{C}' = \bigcup_{i=1}^{n} \mathcal{C}_i.$$

Pour se convaincre que $(U, \mathcal{C})$ est satisfaisable si et seulement si $(U', \mathcal{C}')$ est satisfaisable, il suffit de constater que $\mathcal{C}_i$ est satisfaisable si et seulement si c_i est satisfaite. ■

8.5.3 $\mathcal{NP}$-complétude de $3COL$

Théorème 8.9 *Le problème $3COL$ est $\mathcal{NP}$-complet.*

Preuve:

1. $3COL \in \mathcal{NP}$: il suffit de choisir la couleur des sommets et de vérifier que deux sommets voisins ne sont jamais de la même couleur.

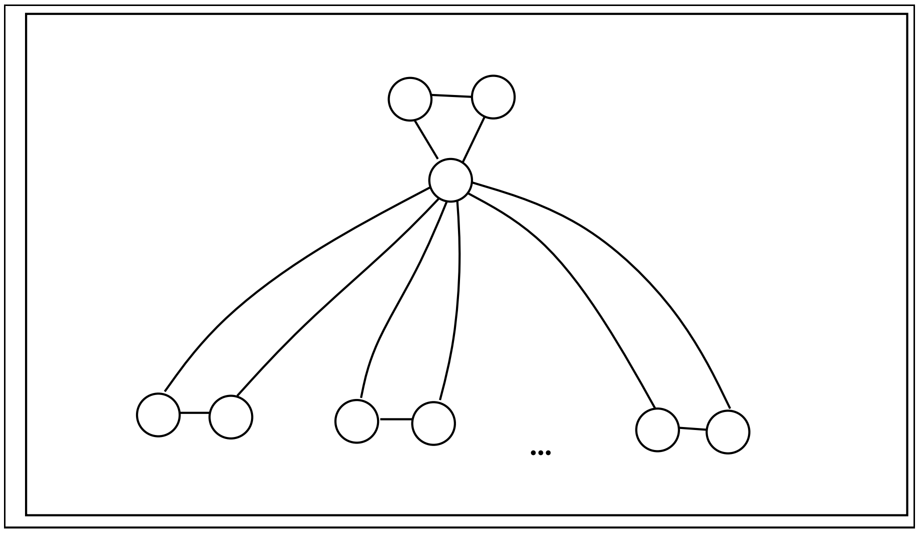

Figure 16: structure du graphe associé aux variables u_is

2. $3SAT \leq_{\mathcal{P}} 3COL$:
Soient U et $\mathcal{C}$, les données d'un problème de $3SAT$. On décrit G constituant un problème de $3COL$ équivalent. Posons, $U = \{u_1, u_2, ..., u_m\}$ et $\mathcal{C} = \{c_1, c_2, ..., c_n\}$. On associe d'abord le sous-graphe de la figure 16 aux variables de U, puis pour chaque clause c_i on aura un sous-graphe G_i comme celui de la figure 17. Pour se convaincre que G est

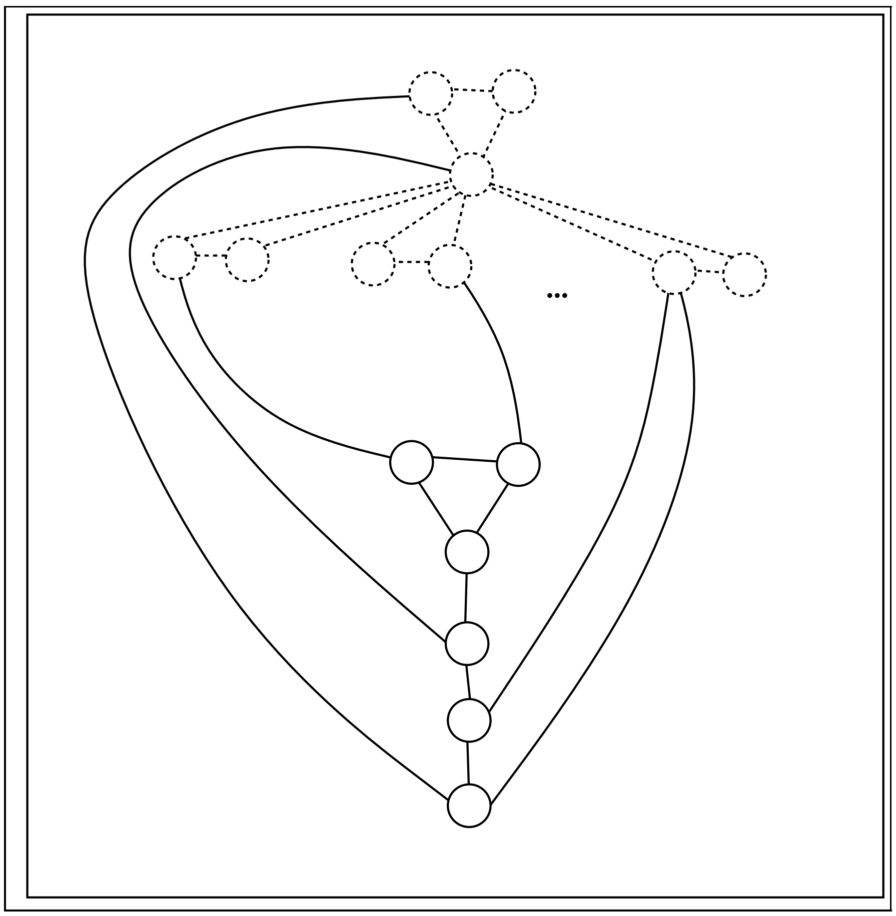

Figure 17: structure du graphe associé aux clauses c_is

3-coloriable si et seulement si $(U, \mathcal{C})$ est satisfaisable, il suffit de constater que G_i est 3-coloriable si et seulement si c_i est satisfaite. ■