

3. Thèse de Church-Turing

Bruno Grenet

Université Grenoble Alpes – IM²AG

L3 Informatique

UE Modèles de calcul – Machines de Turing



<https://membres-ljk.imag.fr/Bruno.Grenet/MCAL-MT.html>

Introduction

Thèse de Church-Turing

Tout ce qui peut être *effectivement calculé* peut l'être par une machine de Turing

Signification

- ▶ *Calcul effectif*: algorithme au sens intuitif
 - ▶ Tout calcul qu'un humain peut faire (en théorie) version *psychologique*
 - ▶ Tout calcul réalisable par un procédé physique version *physique*
- ▶ *Thèse* au sens d'hypothèse :
 - ▶ Par nature indémontrable mathématiquement
 - ▶ Mais réfutable : si on trouve par exemple un procédé physique non simulable par une machine de Turing

⇒ La notion *intuitive* de calculabilité est *capturée* par le modèle de machine de Turing

Contents

1. Reconnaître, décider, calculer
2. Machines RAM et langage WHILE
3. Autres modèles

Contents

1. Reconnaître, décider, calculer

2. Machines RAM et langage WHILE

3. Autres modèles

Fonctions et langages

Définitions

Soit Σ un alphabet fini et Σ^* les mots sur Σ

► Fonction *partielle* $f : \Sigma^* \rightarrow \Sigma^*$: pour tout $w \in \Sigma^*$,

► soit $f(w) = w' \in \Sigma^*$

► soit $f(w)$ est *indéfini*, noté $f(w) \uparrow$

$f(w)$ défini : $f(w) \downarrow$

► Fonction *totale* $f : f(w)$ est définie pour tout w

► Langage $L \subset \Sigma^*$: sous-ensemble de Σ^*

Liens

► Langage associé à une fonction (partielle) $f : L_f = \{w : f(w) \in \Sigma^*\}$ *domaine*

► Fonction caractéristique de $L : \chi_L : \Sigma^* \rightarrow \{0, 1\}$ définie par $\chi_L(w) = \begin{cases} 1 & \text{si } w \in L \\ 0 & \text{sinon} \end{cases}$

► L et χ_L sont le *même objet*

$$L = \{w \in \Sigma^* : \chi_L(w) = 1\}$$

Remarque

► Un algorithme calcule une fonction partielle

Rappel : codage binaire

Toute donnée peut se coder en binaire $\rightarrow \Sigma = \{0, 1\}$

Rien de nouveau !

- ▶ Entiers : écriture binaire ; complément à deux pour les négatifs
- ▶ Flottants : *conventions* sur 32 bits (float), 64 bits (double), ...
- ▶ Chaînes de caractères : codage ASCII, UTF-8, ...
- ▶ ...

Codages de codage

- ▶ Paire (a, b) : par ex. $\langle a \rangle \# \langle b \rangle$ puis codage de $\{0, 1, \#\}$ dans $\{0, 1\}$
- ▶ Triplet (a, b, c) : par ex. $\langle \langle a, b \rangle, c \rangle$
- ▶ Listes, matrices, graphes, fonctions $\{0, 1\}^8 \rightarrow \{0, 1\}, \dots$

Conclusions

- ▶ Codage : composition de codages simples
- ▶ *Sémantique* (sens) d'un mot $w \in \{0, 1\}^*$: *convention*

Reconnaître, décider, calculer

Définitions informelles

- ▶ Un algorithme $\mathcal{A}$ **reconnaît** un langage L si $\mathcal{A}(w) = 1 \iff w \in L$
- ▶ Un algorithme $\mathcal{A}$ **décide** un langage L si pour tout w , $\mathcal{A}(w) = \begin{cases} 1 & \text{si } w \in L \\ 0 & \text{si } w \notin L \end{cases}$
- ▶ Un algorithme $\mathcal{A}$ **calcule** une fonction f si pour tout w , $\mathcal{A}(w) = f(w)$

Remarques

- ▶ Reconnaissance : pour $w \notin L$, $\mathcal{A}$ peut répondre tout sauf 1, ou boucler
- ▶ Calcul :
 - ▶ $\mathcal{A}(w)$ termine si $w \in L_f$ et boucle si $w \notin L_f$
 - ▶ Si f est totale, $\mathcal{A}$ doit terminer sur toute entrée

Lemme

Un algorithme $\mathcal{A}$ décide un langage L si et seulement si $\mathcal{A}$ calcule χ_L

$$\mathcal{A} \text{ décide } L \iff \mathcal{A}(w) = \begin{cases} 1 & \text{si } w \in L \\ 0 & \text{si } w \notin L \end{cases} = \chi_L(w)$$

Conséquence de la thèse de Church-Turing

Définitions

- ▶ Un langage $L \subset \Sigma^*$ est
 - ▶ reconnaissable s'il existe une machine de Turing $\mathcal{M}$ t.q. $L = L(\mathcal{M})$
 - ▶ décidable s'il existe une machine de Turing $\mathcal{M}$ t.q. $\chi_L = f_{\mathcal{M}}$
- ▶ Une fonction $f : \Sigma^* \rightarrow \Sigma^*$ est
 - ▶ calculable s'il existe une machine de Turing $\mathcal{M}$ t.q. $f = f_{\mathcal{M}}$

On se répète ?

- ▶ Page précédente : définitions informelles basées sur la notion intuitive d'algorithme
- ▶ Page courante : définitions formelles basées sur les machines de Turing
- ▶ Justification : thèse de Church-Turing

Remarques

- ▶ Définitions heureusement cohérentes avec celles du cours 1!

Différence entre reconnaître et décider

$$L_\pi = \{w : w \text{ apparaît dans l'écriture binaire de } \pi\}$$

Langage reconnaissable

- ▶ Algorithme :
 1. Calculer l'écriture binaire (infinie !) de π
 2. À chaque nouveau bit calculé, vérifier si w apparaît dans ce qu'on a déjà calculé
- ▶ Analyse :
 - ▶ Si w apparaît : l'algorithme peut renvoyer 1
 - ▶ Si w n'apparaît pas : l'algorithme boucle

Est-il décidable ?

- ▶ Conjecture (non prouvée) : tout mot $w \in \{0,1\}^*$ apparaît dans l'écriture binaire de π
 - ▶ si la conjecture est vraie, l'algorithme renvoie toujours 1
 - ▶ autrement dit, $L_\pi = \{0,1\}^*$ et est clairement décidable !
- ▶ Sans la conjecture : je ne sais pas !

Il semble que personne ne sait

Contents

1. Reconnaître, décider, calculer

2. Machines RAM et langage WHILE

3. Autres modèles

Introduction

Motivation

- ▶ Thèse de Church-Turing : algorithme = machine de Turing
- ▶ Machines de Turing très éloignées de la pratique

Objectif et méthode

- ▶ Définir un mini-langage de programmation
 - ▶ proche des langages habituels
 - ▶ *équivalent* aux machines de Turing
- ▶ Utilisation du formalisme des machines RAM

WHILE

Machine RAM

- ▶ RAM = *Random Access Memory*
- ▶ Machine à registres, opérations *élémentaires* sur les registres
- ▶ Proche des processeurs et de l'assembleur

Machine RAM

Description

Une machine RAM est constituée

- ▶ de registres $R_0, R_1, \dots$ contenant chacun un mot $w \in \{0, 1\}^n$ $n =$ taille de mot
- ▶ d'une suite finie numérotée d'instructions, de cinq types possibles :
 - $\text{COPY}(R_i, R_j)$: copie le registre j dans le registre i
 - $\text{INC}(R_i)$: incrémente le registre i (*modulo* 2^n)
 - $\text{DEC}(R_i)$: décrémente le registre i (*modulo* 2^n)
 - $\text{JUMP}(R_i, \ell)$: si $R_i = 0$, aller à l'instruction ℓ ; sinon continuer
 - STOP : arrêt du programme

Fonctionnement

- ▶ Initialement, la 1^{ère} instruction est exécutée
- ▶ On suit les instructions dans l'ordre, sauf si JUMP ou STOP
- ▶ Entrée dans R_0 (et suivants si plusieurs entrées), autres registres à 0
- ▶ Sortie dans R_0 (et suivants si plusieurs sorties)

Exemples

Addition

Entrées: x dans R_0 ; y dans R_1

Sortie: $x + y \bmod 2^n$

1. JUMP($R_1, 5$)
2. INC(R_0)
3. DEC(R_1)
4. JUMP($R_2, 1$)
5. STOP

R_0	R_1
5	3
6	2
7	1
8	0

Test d'égalité

Entrées: x dans R_0 ; y dans R_1

Sortie: 1 si $x = y$; 0 sinon

1. JUMP($R_0, 5$)
2. DEC(R_0)
3. DEC(R_1)
4. JUMP($R_2, 1$)
5. JUMP($R_1, 7$)
6. STOP
7. INC(R_0)
8. STOP

R_0	R_1
2	2
1	1
0	0
1	0

R_0	R_1
2	1
1	0
0	$2^n - 1$

Variantes possibles

Jeu d'instructions *enrichi*

- ▶ Par exemple : ADD, SUB, MUL, EQUAL, ...
- ▶ Équivalence : puisqu'on sait implanter ADD, on peut l'ajouter ; etc.

Jeu d'instructions réduit

- ▶ INC, DEC, JUMP uniquement sur $R_0 \rightarrow$ utiliser COPY
- ▶ COPY remplacé par $\text{LOAD}(R_i) = \text{COPY}(R_0, R_i)$ et $\text{STORE}(R_i) = \text{COPY}(R_i, R_0)$

Preuve de l'équivalence

$\text{INC}(R_4) \rightarrow$

$\text{STORE}(R_{12})$	
$\text{LOAD}(R_4)$	
$\text{INC}(R_0)$	
$\text{STORE}(R_4)$	
$\text{LOAD}(R_{12})$	

, etc...

Simulation d'une machine RAM par une machine de Turing

Machine à simuler

- ▶ On suppose la version *réduite*: calculs uniquement sur R_0 , LOAD/STORE au lieu de COPY

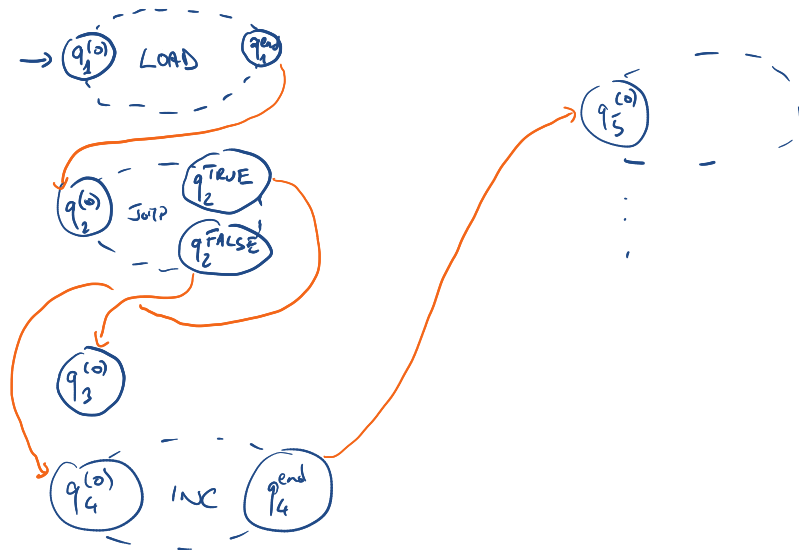
Description de la machine

- ▶ Ruban : $R_0 \# R_1 \# \dots \# R_{\max}$
- ▶ Instruction numéro ℓ : ensemble d'états $q_\ell^{(0)}, q_\ell^{(1)}, q_\ell^{(2)}, \dots$ (état initial : $q_1^{(0)}$)
- ▶ $\Sigma = \{0, 1, \#, \square\}$

Simulation de l'instruction ℓ

- ▶ Début de la simulation : état $q_\ell^{(0)}$ et tête au début du ruban
- ▶ Instructions :
 - ▶ INC/DEC : incrément ou décrétement de R_0 et passage dans l'état $q_{\ell+1}^{(0)}$
 - ▶ JUMP(R_0, m) : vérifier si $R_0 = 0^n$ et passer soit dans l'état $q_m^{(0)}$, soit $q_{\ell+1}^{(0)}$
 - ▶ LOAD(R_i) et STORE(R_i) :
 - ▶ passer i symboles $\#$ pour atteindre R_i
 - ▶ Copie dans un sens ou dans l'autre
 - ▶ STOP : état $q_\ell^{(0)}$ sans transition

Intuition graphique de la construction



1. $LOAD(R_3)$
2. $JUMP(R_0, 4)$
3. $STOP$
4. $INC(R_0)$
- ...

Description du langage WHILE

Spécifications

- ▶ Variables $x_0, x_1, \dots$ contenant des entiers non signés sur n bits
 - ▶ $x_i \leftarrow 0$
 - ▶ $x_i \leftarrow x_j$
 - ▶ $x_i \leftarrow x_i \pm 1 \pmod{2^n}$
 - ▶ $x_i \leftarrow x_j \pm x_k \pmod{2^n}$
 - ▶ Structures conditionnelles :
 - ▶ while $x_i \neq 0$: P
 - ▶ for $x_i = 0$ to $x_j - 1$: P
 - ▶ Entrée(s) : dans $x_1, x_2, \dots$
 - ▶ Valeur de retour : dans x_0
- P, Q : suites d'instructions
- ▶ if $x_i > x_j$: P
else: Q

On peut tout faire avec !

- ▶ Autres opérations : algorithmes pour la multiplication, autres comparaisons, ...
- ▶ Autres types de données :
 - ▶ chaînes de caractères, entiers signés $\rightarrow$ convention
 - ▶ flottants $\rightarrow$ deux entiers mantisse et exposant
 - ▶ ...

Exemples de programmes WHILE

FIBONACCI

Entrée: $x_1 \geq 0$

Sortie: $x_0 = F_{x_1}$

1. $x_0 \leftarrow 0; x_2 \leftarrow 0; x_2 \leftarrow x_2 + 1$
2. for $x_4 = 0$ to $x_1 - 1$:
3. $x_3 \leftarrow x_0 + x_2$
4. $x_0 \leftarrow x_2$
5. $x_2 \leftarrow x_3$

$$F_n = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ F_{n-1} + F_{n-2} & \text{sinon} \end{cases}$$

EUCLIDE

Entrées: x_1 et x_2

Sortie: $x_0 = \text{PGCD}(x_1, x_2)$

1. if $x_1 > x_2$: $x_0 \leftarrow x_1; x_3 \leftarrow x_2$
2. else: $x_0 \leftarrow x_2; x_3 \leftarrow x_1$
3. while $x_3 \neq 0$:
4. $x_0 \leftarrow x_0 - x_3$
5. if $x_3 > x_0$:
6. $x_4 \leftarrow x_0; x_0 \leftarrow x_3; x_3 \leftarrow x_4$

Exemples de programmes WHILE

FIBONACCI

Entrée: $n \geq 0$

Sortie: $f = F_n$

1. $f \leftarrow 0; g \leftarrow 0; g \leftarrow g + 1$
2. for $i = 0$ to $n - 1$:
3. $h \leftarrow f + g$
4. $f \leftarrow g$
5. $g \leftarrow h$

$$F_n = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ F_{n-1} + F_{n-2} & \text{sinon} \end{cases}$$

EUCLIDE

Entrées: x et y

Sortie: $a = \text{PGCD}(x, y)$

1. if $x > y$: $a \leftarrow x; b \leftarrow y$
2. else: $a \leftarrow y; b \leftarrow x$
3. while $b \neq 0$:
4. $a \leftarrow a - b$
5. if $b > a$:
6. $t \leftarrow a; a \leftarrow b; b \leftarrow t$

Simplification du langage WHILE

Opérations

► Addition $c \leftarrow a + b$:

1. $c \leftarrow a$; $t \leftarrow b$
2. while $t \neq 0$:
3. $c \leftarrow c + 1$; $t \leftarrow t - 1$

► Soustraction $c \leftarrow a - b$:

1. $c \leftarrow a$; $t \leftarrow b$
2. while $t \neq 0$:
3. $c \leftarrow c - 1$; $t \leftarrow t - 1$

Structures conditionnelles

► for $x_i = 0$ to $x_j - 1$: P

1. $x_i \leftarrow 0$; $t \leftarrow x_j$
2. while $t \neq 0$:
3. P
4. $x_i \leftarrow x_i + 1$; $t \leftarrow t - 1$

► if $x_i > x_j$: P ; else: Q

1. $y_i \leftarrow x_i$; $y_j \leftarrow x_j$; $s \leftarrow 1^*$
2. while $y_i \neq 0$:
3. $t \leftarrow 1^*$; $z_j \leftarrow y_j$
4. while $z_j \neq 0$:
5. $y_i \leftarrow y_i - 1$; $y_j \leftarrow y_j - 1$
6. $z_j \leftarrow 0$; $t \leftarrow 0$
7. while $t \neq 0$:
8. P ; $y_i \leftarrow 0$
9. $s \leftarrow 0$; $t \leftarrow 0$
10. while $s \neq 0$:
11. Q ; $s \leftarrow 0$

Simplification du langage WHILE

Opérations

► Addition $c \leftarrow a + b$:

1. $c \leftarrow a; t \leftarrow b$
2. while $t \neq 0$:
3. $c \leftarrow c + 1; t \leftarrow t - 1$

► if $x_i > x_j$: P ; else: Q

1. $y_i \leftarrow x_i; y_j \leftarrow x_j; s \leftarrow 1^*$
2. while $y_i \neq 0$:
3. $t \leftarrow 1^*; z_i \leftarrow y_i$

Langage WHILE minimal

$x_i \leftarrow 0$

$x_i \leftarrow x_j$

$x_i \leftarrow x_i \pm 1$

while $x_i \neq 0$: P

► for $x_i = 0$ to $x_j = 1$:

1. $x_i \leftarrow 0; x_c \leftarrow x_j$
2. while $t \neq 0$:
3. P
4. $x_i \leftarrow x_i + 1; t \leftarrow t - 1$

11. $Q; s \leftarrow 0$

Compilation pour machine RAM

Objectif: *compiler* un programme WHILE vers l'assembleur RAM

- ▶ Langage WHILE minimal vers machine RAM non réduite
- ▶ Très similaire à la vraie compilation, mais aucun objectif d'optimisation

Algorithme de compilation

- ▶ Variable x_i dans le registre R_i
 - ▶ $x_i \leftarrow x_i \pm 1$: INC/DEC(R_i); $x_i \leftarrow x_j$: COPY(R_i, R_j);
 - ▶ $x_i \leftarrow 0$: COPY(R_i, R_j) où x_j n'est pas utilisée
 - ▶ while $x_i \neq 0$:
 - instruction1
 - instruction2
 - ...
 - instructionK
 - suite
- ℓ . JUMP($R_i, \ell + k + 2$)
 - $\ell + 1$. instruction1
 - ...
 - $\ell + k$. instructionK
 - $\ell + k + 1$. JUMP(R_j, ℓ) (x_j non utilisée)
 - $\ell + k + 2$. suite

Remarque

RAM $\rightarrow$ WHILE possible également

Bilan provisoire

Les machines de Turing peuvent simuler n'importe quel langage de programmation

Arguments

- ▶ Simulation de machine RAM par machine de Turing
- ▶ Compilation de WHILE sur machine RAM
- ▶ Transformation d'un langage quelconque vers WHILE
 - ▶ Types de données : conventions
 - ▶ Opérations de base : quelques algorithmes
 - ▶ Appels récursifs : utilisation de pile d'appel, etc.

non trivial

Et dans l'autre sens

- ▶ Peut-on simuler une machine de Turing sur une machine RAM ?
 - ▶ *A priori* oui → simulateurs dans des langages puis compilation vers RAM
 - ▶ Mais un problème : machine RAM avec mémoire bornée
 - ▶ Suite finie d'instructions → nombre fini de registres utilisés
 - ▶ Registres de n bits

Solution : un registre d'adresse

Machine RAM non bornée

- ▶ Ajout d'un *registre d'adresse* A contenant un entier **non borné**
- ▶ Deux instructions :
 - ▶ $\text{LOAD}(R_0, A)$: si A contient l'entier j , copie R_j dans R_0
 - ▶ $\text{STORE}(R_0, A)$: si A contient l'entier j , copie R_0 dans R_j

Remarques

- ▶ Adressage indirect en assembleur ; tableaux dynamiques
- ▶ Registre d'adresse non borné : peu réaliste mais nécessaire théoriquement

x_{x_i}

Simulations

- ▶ RAM par machine de Turing : déjà fait, sauf
 - ▶ LOAD/STORE : compter le nombre de # qu'on passe pour atteindre R_j
- ▶ Machine de Turing par RAM :
 - ▶ Simulation d'une machine avec ruban borné à gauche
 - ▶ Un registre par case du ruban + registre d'adresse pour la position de la tête
 - ▶ Instructions « if ... : ... else: ... » pour chaque état

Bilan

Modèles théoriques équivalents

- ▶ Machine de Turing (et variantes)
- ▶ Machine RAM avec registre d'adresse non borné (et variantes)
- ▶ Langage WHILE avec adressage indirect (et variantes)

Le lien avec les ordinateurs

- ▶ Modèle le plus proche :
 - ▶ machine RAM avec registre d'adresse borné donc équivalent à un automate fini...
 - ▶ mais modèle non pertinent impossible de reconnaître $a^n b^n$?
- ▶ Pourquoi un modèle non borné ?
 - ▶ en pratique, borne d'adresse gigantesque
 - ▶ en théorie, on pourrait construire une machine non bornée sauf limites physiques...

À retenir

- ▶ Les machines de Turing ou RAM non bornée sont de *bons* modèles des ordinateurs
- ▶ Comme tout modèle, il y a des limites

Contents

1. Reconnaître, décider, calculer

2. Machines RAM et langage WHILE

3. Autres modèles

Retour sur la thèse de Church-Turing

La notion de *calcul effectif* est capturée par le modèle de machine de Turing

Plein de manières de calculer ?

- ▶ Programmation fonctionnelle, λ -calcul
- ▶ Langages de programmation divers et variés
- ▶ Calcul quantique, calcul analogique
- ▶ Automates cellulaires
- ▶ Calcul moléculaire, par ADN, par membrane, ...
- ▶ *Minecraft*, démineur, ...
- ▶ ...

Tous équivalents aux machines de Turing !

- ▶ Preuves plus ou moins compliquées
- ▶ Modèles Turing-complets

Les équivalences historiques

Trois modèles historiques

- ▶ Fonctions μ -récurives
- ▶ λ -calcul
- ▶ Machines de Turing

Herbrand-Gödel (1931-34)

Kleene-Church (1932-36)

Turing (1936)

Équivalences

- ▶ Équivalence λ -calcul / fonctions μ -récurives
- ▶ Équivalence machines de Turing avec les précédents

Kleene-Church (1936)

Turing (1937)

- ▶ Les trois modèles de calculs initiaux prouvés équivalents dès leur invention
- ▶ Point de départ de la thèse de Church-Turing
- ▶ Résultats suivants : renforcement de la thèse

Un exemple : les automates cellulaires

Description

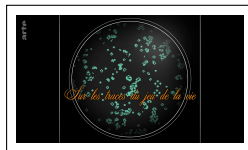
- ▶ Grille infinie de *cellules*, contenant chacune un *symbole* d'un alphabet Σ
- ▶ À chaque étape, évolution synchronisée de toutes les cellules
- ▶ Règle d'évolution basée sur le symbole de la cellule et de ses voisines
- ▶ Bien sûr : des variantes équivalentes !

Qu'est-ce que ça modélise ?

- ▶ Système physique constitué de composants élémentaires en interaction
- ▶ *Agents* en interaction
- ▶ Systèmes dynamiques
- ▶ Parallélisme maximal

Remarques

- ▶ Exemple célèbre : jeu de la vie
- ▶ Aspect intéressant : aspect *local* du calcul



Conclusion

Tous les modèles de calculs (raisonnables) sont équivalents !

Conséquences

- ▶ La notion d'algorithme est *bien définie* → toutes les définitions sont équivalentes
- ▶ Deux caractéristiques importantes du calcul :
 - ▶ mémoire non bornée
 - ▶ localité

Au delà de l'informatique

- ▶ Thèse de Church-Turing physique :
 - ▶ tout *dispositif physique* est simulable par une machine de Turing
- ▶ La notion de *calcul* est une notion fondamentale pour comprendre le monde

Modèle et réalité

- ▶ Problème de la mémoire non bornée → non réaliste (?) mais meilleur modèle
- ▶ Notion de calcul intéressante même sans limites physiques