

### 3. Thèse de Church-Turing

Bruno Grenet

Université Grenoble Alpes – IM<sup>2</sup>AG  
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

# Table des matières

1. Reconnaître, décider, calculer

2. Machines RAM et mini-langage de programmation

3. Autres modèles

# Table des matières

1. Reconnaître, décider, calculer

2. Machines RAM et mini-langage de programmation

3. Autres modèles

# Fonctions et langages

## Définitions

Soit  $\Sigma$  un alphabet fini et  $\Sigma^*$  les mots sur  $\Sigma$

- ▶ Fonction (partielle)  $f : \Sigma^* \rightarrow \Sigma^*$  : pour tout  $w \in \Sigma^*$ ,
  - ▶ soit  $f(w) = w' \in \Sigma^*$
  - ▶ soit  $f(w)$  est indéfinie, noté  $f(w) = \perp$
- ▶ Fonction totale  $f : f(w)$  est définie pour tout  $w$
- ▶ Langage  $L \subset \Sigma^*$  : sous-ensemble de  $\Sigma^*$

Addition  $(a, b)$  :

Tant que  $b \neq 0$  :

$$a \leftarrow a+1$$

$$b \leftarrow b-1$$

Renvoyer  $a$

Attention : notation et pas symbole !

## 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  $\chi_L$  sont le même objet

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

## Remarques

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

## Lemme

Un algorithme  $A$  décide un langage  $L$  si et seulement si  $A$  calcule  $\chi_L$

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

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

# 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  $\pi$
  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  $\pi$ 
  - ▶ 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*

# Table des matières

1. Reconnaître, décider, calculer

2. Machines RAM et mini-langage de programmation

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

## 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  $\ell$  ; sinon continuer
  - $\text{STOP}$  : arrêt du programme

## Fonctionnement

- ▶ Initialement, la 1<sup>ère</sup> 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
<span style="border: 1px solid black; padding: 2px;">8</span>	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	1
1	0
<span style="border: 1px solid black; padding: 2px;">0</span>	$2^n - 1$

$R_0$	$R_1$
3	4
2	3
1	2
<span style="border: 1px solid black; padding: 2px;">0</span>	1
<hr/>	
2	2
1	1
0	0
<span style="border: 1px solid black; padding: 2px;">1</span>	0

# 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

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

- $\text{COPY}(R_{12}, R_0)$
- $\text{COPY}(R_0, R_4)$
- $\text{INC}(R_0)$
- $\text{COPY}(R_4, R_0)$
- $\text{COPY}(R_0, R_{12})$

$\text{COPY}(R_i, R_j) \rightarrow$

- $\text{STORE}(R_{12})$
- $\text{LOAD}(R_j)$
- $\text{STORE}(R_i)$
- $\text{LOAD}(R_{12})$

*n'importe quel registre libre*

# 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

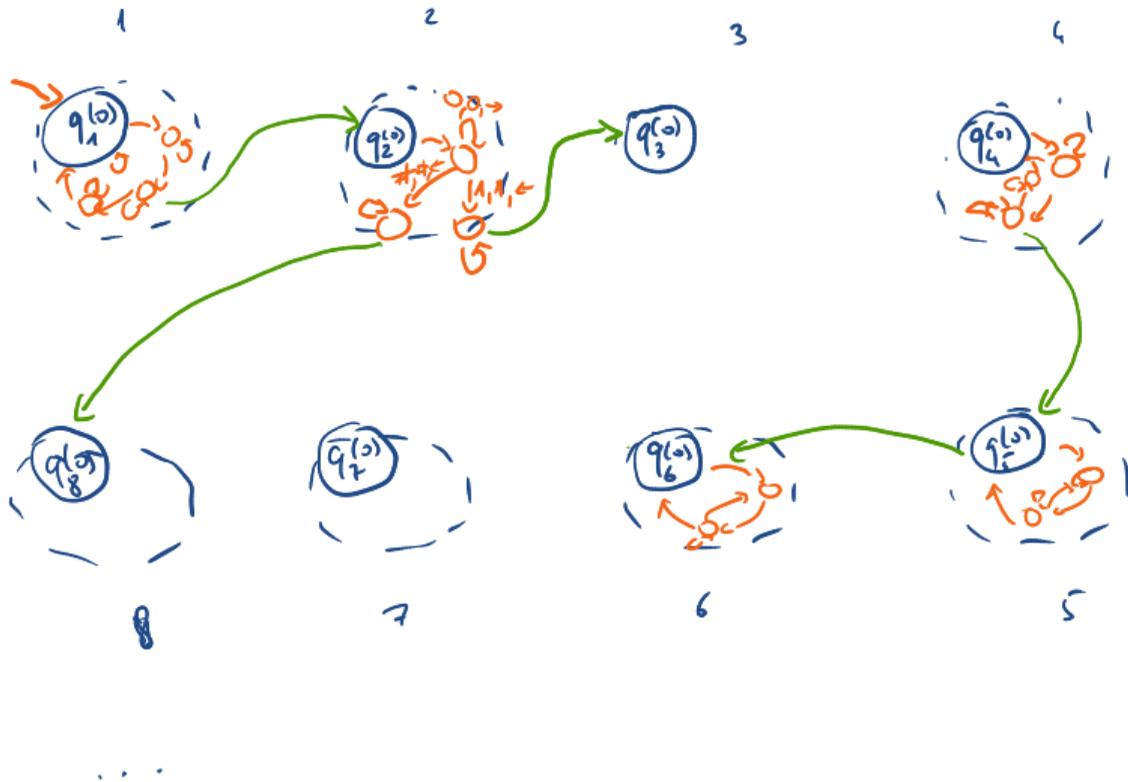
- ▶ Deux rubans : ruban des registres contenant  $\langle 0 \rangle; R_0 \# \langle 1 \rangle; R_1 \# \dots$  et ruban de travail
- ▶ Pour l'instruction numéro  $\ell$ , plusieurs états  $q_\ell^{(0)}, q_\ell^{(1)}, q_\ell^{(2)}, \dots$  (état initial :  $q_1^{(0)}$ )
- ▶  $\Sigma = \{0, 1, ;, \#, \square\}$

## Simulation de l'instruction $\ell$

- ▶ 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( $R_i$ ) :
    - ▶ écrire  $\langle i \rangle$  sur le ruban de travail
    - ▶ parcourir le ruban des registres jusqu'à  $\langle i \rangle$  et effectuer la copie grâce au ruban de travail
  - ▶ STOP : état  $q_\ell^{(0)}$  sans transition

# Intuition graphique de la construction

1. STORE( $R_i$ )
2. JUMP(8)
3. STOP
4. INC
5. LOAD( $R_j$ )
- ...



# Un mini-langage de programmation

## Spécifications

- ▶ Type de données :
  - ▶ variable  $x$  : entier non signé sur  $n$  bits
  - ▶ variable  $T_{[N]}$  : tableau de  $N$  entiers non signés sur  $n$  bits
- ▶ Opérations : incrément et décrétement de variables ; affectations ;
- ▶ Structures conditionnelles : « Tant que » ; « Si ... alors ... sinon »
- ▶ Entrée(s) et valeur(s) de retour

## On peut tout faire avec !

- ▶ Autres opérations : algorithmes pour l'addition, multiplication, comparaisons ...
- ▶ Boucle « Pour » : utiliser « Tant que »
- ▶ Autres types de données :
  - ▶ chaînes de caractères, entiers signés → convention
  - ▶ flottants → deux entiers mantisse et exposant
  - ▶ ...

# Exemple : algorithme d'Euclide

Entrées :  $x$  et  $y$

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

1.  $a \leftarrow x ; b \leftarrow y$
2. Si  $x < y : a \leftarrow y ; b \leftarrow x$
3. Tant que  $b \neq 0$  :
4.   Tant que  $b \leq a$  :
5.        $a \leftarrow a - b$
6.    $t \leftarrow a ; a \leftarrow b ; b \leftarrow t$
7. Renvoyer  $a$

## À définir

- ▶ Soustraction  $a - b$  :
  - ▶ Par incrément / décrétement comme l'addition
  - ▶ Copie de  $a$  et  $b$  dans des variables temporaires
- ▶ Comparaison  $x < y$  :
  - ▶ Transformer en  $z > 0$  avec  $z = y - x$

## Remarques

- ▶ On ramène tout à incrément, décrétement et test à zéro
- ▶ On peut aussi remplacer « Si test alors cas\_oui sinon cas\_non »
  1.  $x \leftarrow 0$  # nouvelle variable
  2. Tant que  $x = 0$  et test : cas\_oui ;  $x \leftarrow 1$
  3. Tant que  $x = 0$  et test\_faux : cas\_non

# Compilation pour machine RAM

## Objectif

- ▶ *Compiler* un programme de notre mini-langage vers l'assembleur RAM
- ▶ Langage simplifié mais machine RAM non réduite
- ▶ Très similaire à la vraie compilation, mais aucun objectif d'optimisation

## Méthode

- ▶ Numérotation des variables dans l'ordre d'apparition
  - ▶  $i^{\text{ème}}$  variable en registre  $R_i$
  - ▶ si tableau : équivalent à  $N$  variables
- ▶ Incrément, décrétement, affectation  $\rightarrow$  INC, DEC, COPY
- ▶ « Renvoyer  $x$  » : copier  $x$  dans  $R_0$ , puis STOP

- ▶ Tant que  $x > 0$  :

instruction1  
instruction2  
...  
instructionK

*suite*

$l$ . JUMP( $R_i, l + k + 2$ )  
 $l + 1$ . instruction1  
...  
 $l + k$ . instructionK  
 $l + k + 1$ . JUMP( $l$ ) (non conditionnel)  
 $l + k + 2$ . *suite*

# Bilan provisoire

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

## Arguments

- ▶ Simulation de machine RAM par machine de Turing
- ▶ Compilation du mini-langage sur machine RAM
- ▶ Transformation d'un langage quelconque vers le mini-langage
  - ▶ 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
  - ▶ Remarque : tableaux uniquement *statiques* dans le mini-langage

# 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

## Simulations

- ▶ RAM par machine de Turing : déjà fait !
- ▶ 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
  - ▶ Suite d'instructions groupées pour chaque état

# Bilan

## Modèles théoriques équivalents

- ▶ Machine de Turing et ses variantes
- ▶ Machine RAM avec registre d'adresse non borné et ses variantes
  - ▶ registres non bornés, séparation registres/mémoire, ...

## 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

# Table des matières

1. Reconnaître, décider, calculer

2. Machines RAM et mini-langage de programmation

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,  $\lambda$ -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  $\mu$ -récurives
- ▶  $\lambda$ -calcul
- ▶ Machines de Turing

Herbrand-Gödel (1931-34)

Kleene-Church (1932-36)

Turing (1936)

## Équivalences

- ▶ Équivalence  $\lambda$ -calcul / fonctions  $\mu$ -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

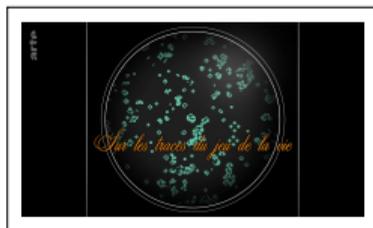
- ▶ Grille infinie de *cellules*, contenant chacune un *symbole* d'un alphabet  $\Sigma$
- ▶ À chaque étape, évolution synchronisée de toutes les cellules
- ▶ Règle d'évolution basée sur le symbole de la cellule et de ses voisines
- ▶ Bien sûr : des variantes équivalentes !

## 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