Chapitre 5

Quelques modèles séquentiels, et leur équivalence

Nous avons maintenant défini ce qu'était un algorithme. La notion d'algorithme au sens précédent est de très haut niveau par rapport à la notion basée sur les machines de Turing, ou sur d'autres langages bas niveau, comme celle qui est utilisée dans la plupart des ouvrages de calculabilité ou de complexité.

Nous ne pouvons éviter de montrer que notre notion d'algorithme correspond à la notion usuelle. L'objet de ce chapitre est de définir quelques modèles et de montrer qu'ils sont équivalents à notre notion d'algorithme.

5.1 Machines de Turing

Revenons sur le modèle classique de la machine de Turing.

5.1.1 Description

Une machine de Turing (déterministe) à k rubans est composée des éléments suivants :

1. Une mémoire infinie sous forme de k rubans. Chaque ruban est divisé en cases. Chaque case peut contenir un élément d'un ensemble M (qui se veut un alphabet). On suppose dans cette section que l'alphabet M est un ensemble fini.
2. *k têtes de lecture* : chaque tête se déplace sur l'un des k rubans.
3. Un programme donné par une *fonction de transition* qui pour chaque état de la machine, précise selon le symbole sous l'une des têtes de lecture,
   (a) l'état suivant ;

(b) le nouvel élément de M à écrire à la place de l'élément de M sous la tête de lecture d'un des rubans ;

(c) un sens de déplacement pour la tête de lecture de ce dernier ruban.

L'exécution d'une machine de Turing peut alors se décrire comme suit : initialement, l'entrée se trouve sur le début du premier ruban. Les cases des rubans qui ne correspondent pas à l'entrée contiennent toutes l'élément **B** (symbole de blanc), qui est un élément particulier de M. A chaque étape de l'exécution, la machine, selon son état, lit le symbole se trouvant sous l'une des têtes de lecture, et selon ce symbole :

- remplace le symbole sous une des têtes de lecture par celui précisé par sa fonction transition ;
- déplace (ou non) cette tête de lecture d'une case vers la droite ou vers la gauche suivant le sens précisé par la fonction de transition ;
- change d'état vers un nouvel état.

5.1.2 Formalisation

La notion de machine de Turing se formalise de la façon suivante :

Définition 5.1 (Machine de Turing) Une machine de Turing à k-rubans *est un 8-uplet*

$$M = (Q, \Sigma, \Gamma, \mathbf{B}, \delta, q_0, q_a, q_r)$$

où

1. Q *est l'ensemble fini des états.*
2. Σ *est un alphabet* fini.
3. Γ *est l'alphabet de travail* fini *:* $\Sigma \subset \Gamma$.
4. $\mathbf{B} \in \Gamma$ *est le caractère blanc.*
5. $q_0 \in Q$ *est l'état initial.*
6. $q_a \in Q$ *est l'état d'acceptation.*
7. $q_r \in Q$ *est l'état de rejet (ou d'arrêt).*
8. δ *est la fonction de transition :* δ *est constitué d'une fonction* δ_1 *de* Q *dans* $\{1, 2, \cdots, k\}$*, et d'une fonction* δ_2 *de* $Q \times \Gamma$ *dans* $Q \times \{1, 2, \cdots, k\} \times \Gamma \times \{\leftarrow, |, \rightarrow\}$.

Le langage accepté par une machine de Turing se définit à l'aide des notions de *configurations* et de dérivations entre configurations.

Formellement : *Une configuration* est donnée par la description des rubans, par les positions des têtes de lecture/écriture, et par l'état interne : une configuration peut se noter $C = (q, u_1 \# v_1, \ldots, u_k \# v_k)$, avec $u_1, \ldots, u_n, v_1, \ldots, v_n \in (\Gamma - \{\mathbf{B}\})^*$, $q \in Q$: u_i et v_i désignent le contenu respectivement à gauche et à droite de la tête de lecture du ruban i, la tête de lecture du ruban i étant sur la première lettre de v_i. Pour simplifier la présentation, on fixe la convention que les mots v_i sont écrits de gauche à droite (la lettre numéro $i+1$ de v_i est à

droite de la lettre numéro i) alors que les u_i sont écrits de droite à gauche (la lettre numéro $i+1$ de u_i est à gauche de la lettre numéro i, la première lettre de u_i étant à gauche de la tête de lecture numéro i).

Une telle configuration est dite *acceptante* si $q = q_a$, *rejetante* si $q = q_r$.

Pour $w \in \Sigma^*$, la configuration initiale correspondante à w est la configuration $C[w] = (q_0, \#w, \#, \ldots, \#)$ (u_1 et les u_i et v_i pour $i \neq 2$ correspondent au mot vide).

On note : $C \vdash C'$ si la configuration C' est le successeur direct de la configuration C par le programme (donné par δ) de la machine de Turing. Formellement, si $C = (q, u_1\#v_1, \ldots, u_k\#v_k)$, et si $a_1, \cdots, a_k$ désignent les premières lettres[1] de $v_1, \cdots, v_k$, et si

$$\delta_1(q) = \ell$$

$$\delta_2(q, a_\ell) = (q', \ell', a', m')$$

alors $C \vdash C'$ si

- $C' = (q', u'_1\#v'_1, \cdots, u'_k\#v'_k)$, et
- pour $i \neq \ell'$, $u'_i = u_i$, $v'_i = v_i$
- si $i = \ell'$,
  - si $m' = |$, $u'_i = u_i$, et v'_i est obtenu en remplaçant la première lettre a_i de v_i par a'.
  - si $m' = \leftarrow$, $v'_i = a'v_i$, et u'_i est obtenu en supprimant la première lettre de u_i.
  - si $m' = \rightarrow$, $u'_i = a'u_i$, et v'_i est obtenu en supprimant la première lettre a_i de v_i.

On appelle *calcul de M sur un mot $w \in \Sigma^*$*, une suite de configurations $(C_i)_{i \in N}$ telle que $C_0 = C[w]$ et pour tout i, $C_i \vdash C_{i+1}$.

Le mot w est dit *accepté* si le calcul sur ce mot est tel qu'il existe un entier t avec C_t acceptante. On dit dans ce cas que w est accepté *en temps t*. Le mot w est dit *refusé* si le calcul sur ce mot est tel qu'il existe un entier t avec C_t rejetante. On dit dans ce cas que w est refusé *en temps t*.

Un langage $L \subset \Sigma^*$ est *dit accepté par M* si pour tout $w \in \Sigma^*$, $w \in L$ si et seulement si w est accepté.

Un langage $L \subset \Sigma^*$ est *dit décidé par M* si pour tout $w \in \Sigma^*$, $w \in L$ si et seulement si w est accepté, et $w \notin L$ si et seulement si w est rejeté.

5.1.3 Une machine de Turing est un algorithme

En fait, cela peut aussi se décrire à un plus haut niveau naturellement comme un algorithme.

Proposition 5.1 *Une machine de Turing à k rubans correspond à un programme du type*

```
ctl_state ←q0
repeat
```

[1] Avec la convention que la première lettre du mot vide est le blanc **B**.

```
  par
 <instructions>
  if ctl_state = q_a then out ←vrai
  if ctl_state = q_r then out ←faux
 endpar
until out!=undef
write out
```

où <instructions> est une suite finie d'instructions du type

```
if ctl_state=q and tape_ℓ(head_ℓ) = a then
par
  tape_ℓ'(head_ℓ') ←a'
  head_ℓ' ←head_ℓ' + m
  ctl_state ←q'
endpar
```

où $q \neq q_a$, $q \neq q_r$, $head_1, \cdots, head_k$ sont des symboles de constantes à valeurs entières, qui codent la position de chacune des têtes de lecture, $tape_1, \cdots, tape_k$ sont des symboles de fonctions d'arité 1, qui codent le contenu de chacun des rubans. Chaque règle de ce type code le fait que $\delta_1(q) = \ell$, $\delta_2(q, a) = (q', \ell', a', m')$, avec m qui vaut respectivement $-1, 0$ ou 1 lorsque m' vaut $\leftarrow, |, \rightarrow$ respectivement.

5.2 Machines de Turing sur une structure $\mathfrak{M}$

Autrement dit, une machine qui travaille sur l'alphabet fini

$$\Sigma \subset \Gamma = \{\gamma_1, \gamma_2, \cdots, \gamma_m\},$$

ne sait écrire que des lettres γ_i, et tester si le symbole sous l'une des têtes de lecture est égal à l'une des lettres γ_i (et déplacer ses têtes de lecture).

Autrement dit, les seules opérations en écriture autorisées sont les constantes $\gamma_1, \gamma_2, \cdots, \gamma_m$, et les seuls tests autorisés correspondent à l'égalité à l'un des symboles γ_i : on peut voir cela comme une machine de Turing sur la signature

$$\mathfrak{M} = (\Gamma, \{\gamma_1, \gamma_2, \cdots, \gamma_m\}, \{\gamma_1?, \gamma_2?, \cdots, \gamma_m?, =\}),$$

où γ_i désigne le symbole de constante qui s'interprète par la lettre γ_i, et $\gamma_i?$ désigne le symbole de prédicat d'arité 1 qui est vrai si et seulement si son argument est la lettre γ_i.

En effet, rien ne nous interdit de considérer des machines de Turing (à k rubans) qui travaillent sur une structure arbitraire plus générale. Une machine qui travaille sur la structure $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$, est autorisée à calculer les fonctions f_i et à tester les relations r_j de la structure. Les cases des rubans contiennent cette fois des éléments de l'ensemble de base M, qui n'est pas nécessairement fini.

5.2.1 Description

La description de ce que l'on appelle une machine de Turing ne change pas, si ce n'est que les rubans contiennent des éléments de M, l'ensemble de base de la structure, possiblement infini, et que la machine est autorisée à effectuer les tests $r_1, \cdots, r_v$ qui correspondent aux relations de la structure, et à effectuer les opérations $f_1, \cdots, f_u$ qui correspondent aux fonctions de la structure.

En d'autres termes, à coup de copier, coller :

Une machine de Turing (déterministe) à k rubans sur la structure

$$\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$$

est composée des éléments suivants.

1. Une mémoire infinie sous forme de k rubans. Chaque ruban est divisé en cases. Chaque case peut contenir un élément de M.
2. *k têtes de lecture* : chaque tête se déplace sur l'un des k rubans.
3. Une *fonction de transition* qui pour chaque état de la machine, et selon le résultat éventuel d'un test de relation de la structure sur l'un des rubans, précise
   (a) l'état suivant ;
   (b) le nouvel élément de M à écrire à la place de l'élément de M sous la tête de lecture d'un des rubans, obtenu comme le résultat de l'application d'une fonction de la structure ;
   (c) un sens de déplacement pour la tête de lecture de ce dernier ruban.

On fixe la convention que l'on lit les arguments d'une relation ou d'une fonction de la structure sur les premières cases à droite de la tête de lecture : si f_i (respectivement r_j) est une fonction (resp. relation) d'arité k, ses arguments sont lus sur les k premiers cases à droite de la tête de lecture.

On fixe aussi les conventions suivantes, de façon à éviter de nombreux problèmes :

1. Les structures considérées contiennent au moins la fonction *id* identité d'arité 1, et la relation *vrai* d'arité 1 qui est toujours vraie : cela permet dans la définition qui suit d'autoriser les machines à recopier le contenu d'une case sur un ruban sur un autre ruban.
2. Les structures considérées contiennent au moins la fonction $\mathbf{0}$ (respectivement : $\mathbf{1}$, $\mathbf{B}$) d'arité 1 qui teste si son argument est $\mathbf{0}$ (resp. $\mathbf{1}$, $\mathbf{B}$) : cela permet de tester si un élément est blanc, par exemple.

Puisque les structures contiennent toujours ces symboles de fonctions et de relations, nous éviterons de les noter explicitement, mais elles sont présentes dans les structures considérées.

5.2.2 Formalisation

Définition 5.2 (Machine de Turing) ***Une machine de Turing à k-rubans*** *sur une structure $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ est un 8-uplet*

$$(Q, \Sigma, \Gamma, \mathbf{B}, \delta, q_0, q_a, q_r)$$

où

1. *Q est l'ensemble fini des états*
2. *Σ est l'alphabet, et correspond à un sous-ensemble de l'ensemble de base de la structure : $\Sigma \subset M$.*
3. *Γ est l'alphabet de travail, et correspond à un sous-ensemble de l'ensemble de base de la structure : $\Sigma \subset \Gamma \subset M$.*
4. *$\mathbf{B} \in \Gamma$ est le caractère blanc.*
5. *$q_0 \in Q$ est l'état initial*
6. *$q_a \in Q$ est l'état d'acceptation*
7. *$q_r \in Q$ est l'état de rejet (ou arrêt).*
8. *δ est la fonction de transition : δ est constitué d'une fonction δ_1 de Q dans $\{1, 2, \cdots, k\} \times \{1, 2, \cdots, v\}$, et d'une fonction δ_2 de $Q \times \{vrai, faux\}$ dans $Q \times \{1, 2, \cdots, k\} \times \{1, 2, \cdots, u\} \times \{\leftarrow, |, \rightarrow\}$.*

Le langage accepté par une machine de Turing se définit toujours à l'aide des notions de *configurations* et de dérivations entre configurations.

On utilise toujours la même convention pour représenter les configurations, et on laisse inchangée la notion de configuration acceptante ou rejetante. Comme auparavant, les mots u_i et v_i sont des mots sur l'alphabet M, mais cette fois cet alphabet est possiblement infini.

On note : $C \vdash C'$ si la configuration C' est le successeur direct de la configuration C par le programme (donné par δ) de la machine de Turing. Formellement, si $C = (q, u_1 \# v_1, \ldots, u_k \# v_k)$, et si

$$\delta_1(q) = (\ell, j)$$

et l'on pose r comme $vrai$ (respectivement $faux$) ssi $r_j(a_{\ell,1}, a_{\ell,2}, \cdots, a_{\ell,k})$ est vrai (resp. faux), où $a_{\ell,1}, a_{\ell,2}, \cdots, a_{\ell,k}$ désignent les k premières lettres du mot v_ℓ, et k est l'arité du symbole de prédicat r_j, et si

$$\delta_2(q, r) = (q', \ell', i, m'),$$

et si l'on pose a' comme l'interprétation de $f_i(a_{\ell,1}, a_{\ell,2}, \cdots, a_{\ell,k'})$ où les symboles $a_{\ell,1}, a_{\ell,2}, \cdots, a_{\ell,k'}$ désignent les k' premières lettres du mot v_ℓ, et k' l'arité de f_i, alors $C \vdash C'$ si

- $C' = (q', u'_1 \# v'_1, \cdots, u'_k \# v'_k)$, et
- pour $i \neq \ell'$, $u'_i = u_i$, $v'_i = v_i$
- si $i = \ell'$,

- si $m = |$, $u'_i = u_i$, et v'_i est obtenu en remplaçant la première lettre a_i de v_i par a'.
- si $m = \leftarrow$, $v'_i = a'v_i$, et u'_i est obtenu en supprimant la première lettre de u_i.
- si $m = \rightarrow$, $u'_i = u_i a'$, et v'_i est obtenu en supprimant la première lettre a_i de v_i.

On laisse inchangée la notion de calcul, de mot accepté, ou refusé.

5.2.3 Une machine de Turing sur une structure $\mathfrak{M}$ est un algorithme sur $\mathfrak{M}$

Proposition 5.2 *Une machine de Turing à k rubans sur la structure $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ correspond à un programme du type*

```
ctl_state ←q0
repeat
   par
 <instructions>
  endpar
    if ctl_state = qa then out ←vrai
    if ctl_state = qr then out ←faux
  endpar
until out!=undef
write out
```

où <instructions> est une suite finie d'instructions du type

```
if ctl_state=q and
     rj(tapeℓ(headℓ), tapeℓ(headℓ + 1), ··· , tapeℓ(headℓ + k)) then
par
   tapeℓ′(headℓ′) ←fi(tapeℓ(headℓ), tapeℓ(headℓ + 1), ··· , tapeℓ(headℓ + k′))
   headℓ′ ←headℓ′ + m
   ctl_state ←q′
endpar
```

ou alors

```
if ctl_state=q and
   not rj(tapeℓ(headℓ), tapeℓ(headℓ + 1), ··· , tapeℓ(headℓ + k)) then
par
   tapeℓ′(headℓ′) ←fi(tapeℓ(headℓ), tapeℓ(headℓ + 1), ··· , tapeℓ(headℓ + k′))
   headℓ′ ←headℓ′ + m
   ctl_state ←q′
endpar
```

où $q \neq q_a$, $q \neq q_r$, $head_1, \cdots, head_k$ sont des symboles de constantes à valeur entières qui codent la position de chacune des têtes de lecture, $tape_1, \cdots, tape_k$ sont des symboles de fonctions d'arité 1, qui codent le contenu de chacun des

rubans. Chaque règle de ce type code le fait que $\delta_1(q) = (\ell, j)$, $\delta_2(q, r) = (q', \ell', i, m')$, *avec* m *qui vaut respectivement* $-1, 0$ *ou* 1 *lorsque* m' *vaut* $\leftarrow, |, \rightarrow$ *respectivement.* k *et* k' *désignent les arités de* r_j *et* f_i.

5.3 Robustesse du modèle

5.3.1 Structures finies comme structures arbitraires

Les définitions précédentes sont conçues précisément pour que le résultat suivant soit vrai :

Proposition 5.3 *Une machine de Turing au sens de la section 5.1, avec alphabet fini* $\Sigma \subset \Gamma = \{\gamma_1, \gamma_2, \cdots, \gamma_m\}$ *correspond à (se simule par) une machine de Turing sur la structure*[2]

$$\mathfrak{M} = (\Gamma, \{\gamma_1, \gamma_2, \cdots, \gamma_m\}, \{\gamma_1?, \gamma_2?, \cdots, \gamma_m?, =\}),$$

où γ_i *désigne le symbole de constante qui s'interprète par la lettre* γ_i, *et* $\gamma_i?$ *désigne le symbole de prédicat d'arité* 1 *qui est vrai si et seulement son argument est la lettre* γ_i.

Démonstration: On remplace chaque instruction de la machine de Turing **if** *ctl_state* $= q$ **and** $tape_\ell(head_\ell) = a$ **then par** $tape_{\ell'}(head_{\ell'}) \leftarrow a' \ldots$ par l'instruction **if** *ctl_state*=q **and** $a?(tape_\ell(head_\ell))$ **then par** $tape_{\ell'}(head_{\ell'}) \leftarrow a' \ldots$ □

Puisque tester si le symbole sous l'une des têtes de lecture γ_i peut aussi se faire en recopiant le symbole sous la tête de lecture sur un autre ruban, en écrivant à sa droite le symbole γ_i et en testant l'égalité, quitte à augmenter le nombre de rubans de 1, on peut aussi affirmer :

Proposition 5.4 *Une machine de Turing au sens de la section 5.1, avec alphabet fini* $\Sigma \subset \Gamma = \{\gamma_1, \gamma_2, \cdots, \gamma_m\}$ *correspond à (se simule par) une machine de Turing sur la structure*

$$\mathfrak{M} = (\Gamma, \{\gamma_1, \gamma_2, \cdots, \gamma_m\}, \{=\}).$$

5.3.2 Des structures finies vers les booléens

On appelle *booléens* la structure

$$\mathfrak{B} = (\{0, 1\}, \mathbf{0}, \mathbf{1}, =).$$

[2]Rappelons nos conventions sur les structures considérée : $\mathfrak{M}$ contient aussi d'autres symboles de fonctions et de relations (comme l'identité, la fonction $vrai$, ...) que nous ne précisons pas pour ne pas alourdir les notations.

Théorème 5.1 *Une machine de Turing sur une structure*

$$\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$$

avec M fini (et donc une machine de Turing au sens de la section 5.1) se simule par une machine de Turing sur la structure

$$\mathfrak{B} = (\{0,1\}, \mathbf{0}, \mathbf{1}, =).$$

Principe: Puisque M est fini, on peut coder chaque élément de M par une suite de 0 et de 1 (son écriture en binaire). Il s'agit alors simplement de construire une machine de Turing qui travaille sur ces écritures en binaire. Puisque M est fini, chaque fonction ou relation ne prend qu'un nombre fini de valeurs : la table de chacune des fonctions et relations peut être stockée dans l'ensemble des états finis Q de la machine de Turing, ou si l'on préfère par une suite finie de **if then else** dans l'algorithme. □

Autrement dit, la calculabilité (les fonctions qui sont calculables) sont les même sur toute structure finie.

Si la structure n'est pas finie (c'est-à-dire si son ensemble de base M n'est pas fini), il n'y aucune raison que la notion de fonction calculable sur la structure et sur les booléens coïncide.

5.3.3 Programmer avec des machines de Turing

La programmation avec des machines de Turing relève de programmation extrêmement bas niveau.

Lemme 5.1 *Fixons deux entiers i et j distincts. On peut écrire un sous-programme d'une machine de Turing à $k \geq \max(i,j)$ rubans qui recopie le contenu à droite de la tête de lecture numéro i sur le ruban numéro j.*

Principe: Le programme consiste à déplacer la tête de lecture numéro j à droite jusqu'à rencontrer un blanc. Puis à déplacer à gauche cette tête de lecture en écrivant à chaque étape sur le ruban numéro j le symbole blanc, de façon à complètement effacer ce ruban, et ramener sa tête de lecture tout à gauche. On déplace alors la tête de lecture numéro i et j case par case vers la droite, en copiant le symbole sous la tête numéro i sur le ruban j, jusqu'à rencontrer un blanc sous la tête numéro i. □

Nous laissons en exercice les problèmes suivants :

Exercice 5.1 *Construire une machine de Turing qui ajoute* 1 *au nombre écrit en binaire (donc avec des* **0** *et* **1***) sur son ruban numéro i.*

Exercice 5.2 *Construire une machine de Turing qui soustrait* 1 *au nombre écrit en binaire (donc avec des* **0** *et* **1***) sur son ruban numéro i.*

Exercice 5.3 *Construire une machine de Turing qui accepte les chaînes de caractère avec le même nombre de* **0** *et de* **1**.

5.4 Machines à $k \geq 2$ piles

5.4.1 Sur une structure finie

Une *machine à k piles*, possède un nombre fini k de piles $r_1, r_2, \cdots, r_k$, qui correspondent à des piles d'éléments de M. Les instructions d'une machine à piles permettent seulement d'empiler un symbole sur l'une des piles, tester la valeur du sommet d'une pile, ou dépiler le symbole au sommet d'une pile.

Si l'on préfère, on peut voir une pile d'éléments de M comme un mot w sur l'alphabet M. Empiler le symbole $a \in M$ correspond à remplacer w par aw. Tester la valeur du sommet d'une pile correspond à tester la première lettre du mot w. Dépiler le symbole au sommet de la pile correspond à supprimer la première lettre de w.

Définition 5.3 *Une machine à k-piles correspond à un programme du type*

```
ctl_state ←0
repeat
   seq
 <instructions>
   if ctl_state = q_a then out ←vrai
   if ctl_state = q_r then out ←faux
   endseq
until out!=undef
write out
```

où <instructions> est une suite finie d'instructions de l'un des types suivants

1. $push_i(a)$
2. pop_i
3. **if** $top_i = a$ **then** $ctl\_state := q$ **else** $ctl\_state := q'$

où $i \in \{1, 2, \cdots, k\}$, a est un symbole d'un alphabet fini Γ, q et q' sont des entiers, et $push_i(a)$, pop_i, top_i désignent respectivement empiler le symbole a sur la pile i, dépiler le sommet de la pile i, et le symbole au sommet de la pile i.

Théorème 5.2 *Toute machine de Turing à k rubans au sens de la section 5.1 peut être simulée par une machine à $2k$ piles.*

Démonstration: Selon la formalisation page 2, une configuration d'une machine de Turing correspond à $C = (q, u_1 \# v_1, \ldots, u_k \# v_k)$, où u_i et v_i désignent le contenu respectivement à gauche et à droite de la tête de lecture du ruban i. On peut voir u_i et v_i comme des piles. Si l'on relit attentivement la formalisation page 2, on observe que les opérations effectuées par le programme de la machine de Turing pour passer de la configuration C a sa configuration successeur C' correspondent à des opérations qui se codent trivialement par des *push*, *pop*, et *top* : on construit donc une machine à $2k$ piles, chaque pile codant l'un des u_i ou v_i (le contenu à droite et à gauche de chacune des têtes de lecture), et qui simule étape par étape la machine de Turing. □

5.4.2 Sur une structure arbitraire

On peut généraliser les machines à k piles sur une structure arbitraire $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$. Une *machine à k piles*, possède un nombre fini k de piles $r_1, r_2, \cdots, r_k$, qui correspondent à des piles d'éléments de l'ensemble de base M. Les instructions d'une machine à piles permettent seulement d'empiler le résultat sur l'une des piles de l'application d'une fonction f_i de la structure à l'une des piles (c'est-à-dire que si f_i est d'arité k, et si $a_1, \cdots, a_k$ désignent les premières lettres d'une des piles, on empile sur l'une des piles $f_i(a_1, \cdots, a_k)$) , tester la valeur d'une relation r_j d'une pile (c'est-à-dire que si r_j est d'arité k, et si $a_1, \cdots, a_k$ désignent les premières lettres d'une des piles, on teste la valeur de $r_j(a_1, \cdots, a_k)$) ou dépiler le symbole au sommet d'une pile.

Selon le même principe que précédemment :

Théorème 5.3 *Toute machine de Turing à k rubans sur une structure arbitraire $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ peut être simulée par une machines à $2k$ piles sur $\mathfrak{M}$.*

5.5 Cas des structures finies : Machines à compteurs

Nous introduisons maintenant un modèle extrêmement rudimentaire : une *machine à compteurs* possède un nombre fini k de compteurs $r_1, r_2, \cdots, r_k$, qui contiennent des entiers naturels. Les instructions d'une machine à compteur permettent seulement de tester l'égalité d'un des compteurs à 0, d'incrémenter un compteur ou de décrémenter un compteur. Tous les compteurs sont initialement nuls, sauf celui codant l'entrée.

Autrement dit :

Définition 5.4 *Une machine à k-compteurs correspond à un programme du type*

```
ctl_state ←0
repeat
   seq
 <instructions>
   if ctl_state = q_a then out ←vrai
   if ctl_state = q_r then out ←faux
   endseq
until out!=undef
write out
```

où <instructions> est une suite finie d'instructions de l'un des types suivants

1. $x_i \leftarrow x_i + 1$
2. $x_i \leftarrow x_i \ominus 1$
3. **if** $x_i = 0$ **then** $ctl\_state := q$ **else** $ctl\_state := q'$

où $i \in \{1, 2, \cdots, k\}$, q *et* q' *sont des entiers, et* $x \ominus 1$ *vaut* $x - 1$ *si* $x \neq 0$, *et* 0 *pour* $x = 0$. *Chacun des registres* x_i *contient un entier naturel.*

Théorème 5.4 *Toute machine à k-piles sur une structure finie (c'est-à-dire sur un alphabet fini) peut être simulée par une machines à* $k + 1$ *compteurs.*

Démonstration: L'idée est de voir une pile $w = a_1 a_2 \cdots a_n$ sur l'alphabet Σ de cardinalité $r - 1$ comme un entier i en base r : sans perte de généralité, on peut voir Σ comme $\Sigma = \{0, 1, \cdots, r-1\}$. Une pile (un mot) w correspond à l'entier $i = a_n r^{n-1} + a_{n-1} r^{n-2} + \cdots a_2 r + a_1$.

On utilise ainsi un compteur i pour chaque pile. Un $k+1$ème compteur (que l'on appellera *compteur supplémentaire*) est utilisé pour ajuster les compteurs et simuler chaque opération (empilement, dépilement, lecture du sommet) sur l'une des piles.

Dépiler correspond à remplacer i par i *div* r. En partant avec le compteur supplémentaire à 0, on décrémente le compteur i de r (en r étapes) et on incrémente le compteur supplémentaire de 1. On répète cette opération jusqu'à ce que le compteur i atteigne 0. On décrémente alors le compteur supplémentaire de 1 en incrémentant le compteur i de 1 jusqu'à ce que le premier soit 0. A ce moment, on lit bien le résultat correct dans le compteur i.

Empiler le symbole a correspond à remplacer i par $i * r + a$. On multiplie d'abord par r : en partant avec le compteur supplémentaire à 0, on décrémente le compteur i de 1 et on incrémente le compteur supplémentaire de r (en r étapes) jusqu'à ce que le compteur i soit à 0. On décrémente alors le compteur supplémentaire de 1 en incrémentant le compteur i de 1 jusqu'à ce que le premier soit 0. A ce moment, on lit $i * r$ dans le compteur i. On incrémente alors le compteur i de a (en a incrémentations).

Lire le sommet d'une pile i correspond à calculer i *mod* r. En partant avec le compteur supplémentaire à 0, on décrémente le compteur i de 1 et on incrémente le compteur supplémentaire de 1. Lorsque le compteur i atteint 0 on s'arrête. On décrémente alors le compteur supplémentaire de 1 en incrémentant le compteur i de 1 jusqu'à ce que le premier soit 0. On fait chacune de ces opérations en comptant en parallèle modulo r dans l'état interne de la machine (ou si l'on préfère en le codant dans l'instruction courante *ctl_state*). □

Observons que nous nous sommes limités ici aux structures finies, car sur une structure infinie, on ne peut pas coder aussi facilement une pile par un entier.

Théorème 5.5 *Toute machine à* $k \geq 3$ *compteurs se simule par une machine à 2 compteurs.*

Démonstration: Supposons d'abord $k = 3$. L'idée est coder trois compteurs i, j et k par l'entier $m = 2^i 3^j 5^k$. L'un des compteurs stocke cet entier. L'autre compteur est utilisé pour faire des multiplications, divisions, calculs modulo m, pour m valant 2, 3, ou 5.

Pour incrémenter i, j ou k de 1, il suffit de multiplier m par 2, 3 ou 5 en utilisant le principe de la preuve précédente.

Pour tester si i, j ou $k = 0$, il suffit de savoir si m est divisible par 2, 3 ou 5, en utilisant le principe de la preuve précédente.

Pour décrémenter i, j ou k de 1, il suffit de diviser m par 2, 3 ou 5 en utilisant le principe de la preuve précédente.

Pour $k > 3$, on utilise le même principe, mais avec les k premiers nombres premiers au lieu de simplement 2, 3, et 5. □

En combinant les résultats précédents, on obtient :

Corollaire 5.1 *Toute machine de Turing (sur un alphabet fini) se simule par une machine à 2 compteurs.*

Observons que la simulation est particulièrement inefficace : la simulation d'un temps t de la machine de Turing nécessite un temps exponentiel pour la machine à 2 compteurs.

5.6 Machines RAM

5.6.1 Introduction

Les *machines RAM (Random Access Machine)* sont des modèles de calcul qui étendent le modèle rudimentaire des machines à compteur, et qui ressemblent plus aux langages machines actuels. Une machine RAM possède des registres qui contiennent des entiers naturels (nuls si pas encore initialisés). Les instructions autorisées dépendent des applications et des ouvrages que l'on consulte, mais elles incluent en général la possibilité de

1. copier le contenu d'un registre dans un autre
2. l'adressage indirect : récupérer/écrire le contenu d'un registre dont le numéro est donné par la valeur d'un autre registre
3. effectuer des opérations élémentaires sur un ou des registres, par exemple additionner 1, soustraire 1 ou tester l'égalité à 0.
4. effectuer d'autres opérations sur un ou des registres, par exemple l'addition, la soustraction, la multiplication, division, les décalages binaires, les opérations binaires bit à bit.

Dans ce qui suit, nous réduirons la discussion aux *SRAM (Successor Random Access Machine)* qui ne possèdent que des instructions des types 1., 2. et 3.

Clairement, tout ce qui peut être fait par une machine à compteurs peut être fait par un SRAM.

Clairement, tout ce qui peut être fait par un SRAM est faisable avec une RAM. La réciproque sera prouvée ultérieurement, puisque nous prouverons que tout ce qui peut être fait par algorithme (et donc par RAM) est faisable par machine de Turing, et donc par une machine à compteur.

5.6.2 Structures finies

Définition 5.5 *Une machine SRAM (Successor Random Access Machine) correspond à un programme du type*

```
ctl_state ←0
repeat
   seq
 <instructions>
   if ctl_state = q_a then out ←vrai
   if ctl_state = q_r then out ←faux
  endseq
until out!=undef
write out
```

où <instructions> est une suite finie d'instructions de l'un des types suivants

1. $x_i \leftarrow 0$
2. $x_i \leftarrow x_i + 1$
3. $x_i \leftarrow x_i \ominus 1$
4. $x_i \leftarrow x_j$
5. $x_i \leftarrow x_{x_j}$
6. $x_{x_i} \leftarrow x_j$
7. ***if*** $x_i = 0$ ***then*** *ctl_state* $\leftarrow j$ ***else*** *ctl_state* $\leftarrow j'$

Chacun des i désigne un entier naturel. Chaque x_i désigne un registre qui contient un entier naturel. x_{x_i} désigne le registre dont le numéro est x_i.

Remarque 5.1 *Si l'on souhaite être puriste, observons qu'il s'agit bien d'un algorithme dans le sens du chapitre 4 : on peut considérer x comme un symbole de fonction d'arité 1. x_i désigne en fait $x(i)$, où i désigne un symbole de constante, dont l'interprétation est fixée à l'entier i. x_{x_i} désigne $x(x(i))$. L'ensemble de base est constitué des entiers naturels.*

Puisqu'une machine à compteurs est une machine SRAM particulière, on obtient :

Corollaire 5.2 *Toute machine de Turing à k rubans se simule par une machine SRAM.*

Par la discussion plus haut, en fait seulement 2 registres suffisent sur les structures finies, si l'on ignore l'efficacité.

5.6.3 Sur une structure arbitraire

On peut généraliser la notion de machine SRAM (ou RAM) aux structures arbitraires

$$\mathfrak{M} = (S, f_1, \cdots, f_u, r_1, \cdots, r_v),$$

et aussi obtenir des simulations plus efficaces même dans le cas des structures finies.

Définition 5.6 *Une machine SRAM (Successor Random Access Machine) sur une structure* $\mathfrak{M} = (S, f_1, \cdots, f_u, r_1, \cdots, r_v)$ *possède deux types de registres : des registres* entiers $x_1, x_2, \cdots, x_n, \cdots$ *et des registres* arbitraires $x'_1, x'_2, \cdots, x'_n, \cdots$. *Les premiers contiennent des entiers naturels. Les autres des valeurs du domaine de base* M *de la structure.*

Le programme de la machine correspond à un programme du type

```
ctl_state ←0
repeat
   seq
 <instructions>
   if ctl_state = q_a then out ←vrai
   if ctl_state = q_r then out ←faux
  endseq
until out!=undef
write out
```

où <instructions> est une suite finie d'instructions de l'un des types suivants

1. $x_i \leftarrow 0$
2. $x_i \leftarrow x_i + 1$
3. $x_i \leftarrow x_i \ominus 1$
4. $x_i \leftarrow x_j$
5. $x_i \leftarrow x_{x_j}$
6. $x_{x_i} \leftarrow x_j$
7. **if** $x_i = 0$ **then** $ctl\_state \leftarrow j$ **else** $ctl\_state := j'$
8. $x'_i \leftarrow x'_j$
9. $x'_i \leftarrow x'_{x_j}$
10. $x'_{x_i} \leftarrow x'_j$
11. **if** $r_j(x'_{\ell_1}, x'_{\ell_2}, \cdots, x'_{\ell_k})$ **then** $ctl\_state \leftarrow j$ **else** $ctl\_state := j'$
12. $x'_i \leftarrow f_i(x'_{\ell_1}, x'_{\ell_2}, \cdots, x'_{\ell_k})$
13. $x'_{x_i} \leftarrow f_i(x'_{\ell_1}, x'_{\ell_2}, \cdots, x'_{\ell_k})$

Chacun des ℓ_i *désigne un entier.* r_j *désigne une des relations de la structure,* f_i *une des fonctions de la structure, et* k *son arité.*

Remarque 5.2 *Comme auparavant, si l'on souhaite être puriste, on peut bien voir cela comme un algorithme dans le sens du chapitre 4, en écrivant* $x(i)$ *pour* x_i, *ou* $x'(i)$ *pour* x'_i, *...etc.*

Théorème 5.6 *Toute machine de Turing sur la structure*

$$\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$$

se simule par une machine SRAM sur $\mathfrak{M}$.

Principe:

Nous traitons le cas où la machine de Turing possède un unique ruban ($k = 1$).

La machine RAM commence par copier l'entrée vers les registres x'_1 à x'_n. Les registres x_1 et x_2 seront utilisés pour la simulation. Le registre x_3 contiendra la longueur du mot sur le ruban de la machine de Turing. On met ensuite la valeur **1** dans le registre x_1 qui contient la position de la tête de lecture. Le registre x_2 code l'état interne de la machine de Turing. A partir de ce moment, la machine RAM simule les étapes de la machine de Turing étape par étape.

Pour simuler l'instruction $\delta_1(q) = (1, j)$, $\delta_2(q, r) = (q', 1, i, m')$, où le symbole de relation r_j est d'arité k, on teste si x_2 vaut q par une suite d'instructions du type $x_3 \leftarrow x_2$ suivi d'une suite de q décrémentations de x_3 en testant s'il est nul à l'issu de cette décrémentation, et bien non nul à chaque étape auparavant.

Si c'est le cas, on charge les symboles à droite de la tête de lecture dans les registres $x'_1, x'_2, \cdots, x'_k$ par des instructions du type $x_2 \leftarrow x_1 \quad x'_1 \leftarrow x'_{x_2} \quad x_2 \leftarrow x_2 + 1$ $x'_2 \leftarrow x'_{x_2} \quad x_2 \leftarrow \quad x_2 + 1 \quad \ldots \; x'_k \leftarrow x'_{x_2}$.

On teste alors la relation r_j par une instruction du type **if** $r_j(x'_1, x'_2, \cdots, x'_k)$ **then** *ctl_state* $\leftarrow q$, où l'instruction q correspond au code à effectuer : c'est-à-dire à écrire dans un premier temps $f_i(x'_1, x'_2, \cdots, x'_k)$ sous la tête de lecture. Pour cela on commence par charger les symboles à droite de la tête de lecture si l'arité de f_i est supérieure à k. On calcule $f_i(x'_1, x'_2, \cdots, x'_k)$ par $x'_1 := f_i(x'_1, x'_2, \cdots, x'_k)$. On met le résultat en place par $x'_{x_1} := x'_1$. On déplace ensuite la tête de lecture en incrémentant ou décrémentant (ou préservant) x_1 selon la valeur de m'. On change l'état interne par une instruction du type $x_2 := 0$ suivi de q incrémentations de x_2.

Si la machine de Turing possède k rubans, on généralise la construction en stockant la case numéro i du ruban ℓ dans le registre x'_i avec $i = k * (q - 1) + \ell$.

Nous laissons à notre lecteur le soin de compléter précisément tous les détails.
□

5.6.4 Équivalence avec les machines de Turing

Nous allons montrer que toute machine RAM peut être simulée par une machine de Turing.

Nous allons en fait prouver dans la section suivante que tout algorithme, et donc toute machine RAM sur une structure arbitraire peut être simulée par machine de Turing.

Pour aider à comprendre la construction très générale de la section qui suit, nous allons montrer dans un premier temps que les machines RAM sur les structures finies, c'est-à-dire au sens de la définition 5.5, peuvent être simulées par les machines de Turing.

Pour cela, nous allons réduire le nombre d'instructions des machines RAM à un ensemble réduit d'instructions (*RISC reduced instruction set* en anglais) en utilisant un unique registre x_0 comme accumulateur.

Définition 5.7 *Une machine RISC (sur une structure finie) est une machine SRAM dont les instructions sont uniquement de la forme*

1. $x_0 \leftarrow 0$
2. $x_0 \leftarrow x_0 + 1$
3. $x_0 \leftarrow x_0 \ominus 1$
4. ***if*** $x_0 = 0$ ***then*** $ctl\_state \leftarrow j$
5. $x_0 \leftarrow x_i$
6. $x_i \leftarrow x_0$
7. $x_0 \leftarrow x_{x_i}$
8. $x_{x_0} \leftarrow x_i$

Clairement, tout programme RAM (au sens de la définition 5.5, c'est-à-dire sur une structure finie) peut être converti en un programme RISC équivalent, en passant systématiquement par l'accumulateur x_0.

Théorème 5.7 *Toute machine RISC peut être simulée par une machine de Turing.*

Démonstration: La machine de Turing qui simule la machine RISC possède 4 rubans. Les deux premiers rubans codent les couples (i, x_i) pour x_i non nul. Le troisième ruban code l'accumulateur x_0 et le quatrième est un ruban de travail.

Plus concrètement, le premier ruban code un mot de la forme

$$\ldots \mathbf{BB} < i_o > \mathbf{B} < i_1 > \cdots \mathbf{B} \cdots < i_k > \mathbf{BB} \cdots .$$

Le second ruban code un mot de la forme

$$\ldots \mathbf{BB} < x_{i_o} > \mathbf{B} < x_{i_1} > \cdots \mathbf{B} \cdots < x_{i_k} > \mathbf{BB} \cdots$$

Les têtes de lecture des deux premiers rubans sont sur le deuxième **B**. Le troisième ruban code $< x_0 >$, la tête de lecture étant tout à gauche. On appelle *position standard* une telle position des têtes de lecture.

Au départ, la donnée du programme RAM est copié sur le second ruban, et **0** est placé sur le ruban 1 signifiant que x_0 contient la donnée du programme.

La simulation est décrite pour trois exemples. Notre lecteur pourra compléter le reste.

1. $x_0 \leftarrow x_0 + 1$: on déplace la tête de lecture du ruban 3 tout à droite jusqu'à atteindre un symbole **B**. On se déplace alors d'une case vers la gauche, et on remplace les **1** par des **0**, en se déplaçant vers la gauche tant que possible. Lorsqu'un **0** ou un **B** est trouvé, on le change en **1** et on se déplace à gauche pour revenir en position standard.

2. $x_{23} \leftarrow x_0$: on parcourt les rubans 1 et 2 vers la droite, bloc délimité par **B** par bloc, jusqu'à atteindre la fin du ruban 1, ou ce que l'on lise un bloc **B10111B** (**10111** correspond à 23 en binaire).

   Si la fin du ruban 1 a été atteinte, alors l'emplacement 23 n'a jamais été vu auparavant. On l'ajoute en écrivant **10111** à la fin du ruban 1, et on recopie le ruban 3 (la valeur de x_0) sur le ruban 2. On retourne alors en position standard.

   Sinon, c'est que l'on a trouvé **B10111B** sur le ruban 1. On lit alors $< x_{23} >$ sur le ruban 2. Dans ce cas, il doit être modifié. On fait cela de la façon suivante :

   (a) On copie le contenu à droite de la tête de lecture numéro 1 sur le ruban 4.

   (b) On copie le contenu du ruban 3 (la valeur de x_0) à la place de x_{23} sur le ruban 2.

   (c) On écrit **B**, et on recopie le contenu du ruban 4 à droite de la tête de lecture du ruban 2, de façon à restaurer le reste du ruban 2.

   (d) On retourne en position standard.

3. $x_0 \leftarrow x_{x_{23}}$: En partant de la gauche des rubans 1 et 2, on parcourt les rubans 1 et 2 vers la droite, bloc délimité par **B** par bloc, jusqu'à atteindre la fin du ruban 1, ou ce que l'on lise un bloc **B10111B** (**10111** correspond à 23 en binaire).

   Si la fin du ruban 1 a été atteinte, on ne fait rien, puisque x_{23} vaut 0 et le ruban 3 contient déjà $< x_0 >$.

   Sinon, c'est que l'on a trouvé **B10111B** sur le ruban 1. On lit alors $< x_{23} >$ sur le ruban 2, que l'on recopie sur le ruban 4. Comme ci-dessus, on parcours les rubans 1 et 2 en parallèle jusqu'à trouver $\mathbf{B} < x_{23} > \mathbf{B}$ où atteindre la fin du ruban 1. Si la fin du ruban 1 est atteinte, alors on écrit **0** sur le ruban 3, puisque $x_{x_{23}} = x_0$. Sinon, on copie le bloc correspondant sur le ruban 1 sur le ruban 3, puisque le bloc sur le ruban 2 contient $x_{x_{23}}$, et on retourne en position standard.

□

En fait, on observera que toute opération $x_0 \leftarrow x_0$ ''opération'' x_i peut être simulée ainsi, dès que ''opération'' correspond à une opération calculable.

5.7 Équivalence entre algorithmes et machines de Turing

Nous sommes prêts à montrer l'équivalence entre notre notion d'algorithme et la notion de machines de Turing sur une structure arbitraire.

Nous avons déjà vu qu'un programme de machine de Turing sur une structure arbitraire $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ correspond à un algorithme sur $\mathfrak{M}$ (étendue par des symboles de fonctions dynamiques).

Il nous reste la partie délicate : montrer la réciproque.

Théorème 5.8 *Tout algorithme sur une structure* $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$ *peut se simuler par machine de Turing sur la structure* $\mathfrak{M}$.

Démonstration: En suivant la discussion du chapitre 4, nous savons qu'à un algorithme est associé un ensemble fini T de termes critiques, tel que l'algorithme puisse se mettre sous la forme normale indiquée par le théorème 4.1.

Pour simuler un tel algorithme, il suffit d'être capable d'évaluer chacun des emplacements $(f, \overline{m})$, pour f un symbole de fonction de la structure. On peut se restreindre aux emplacements utiles, au sens de la définition du chapitre 4 : c'est-à-dire aux symboles f dynamiques, puisque les autres ont une valeur fixée, et avec $[\![(f, \overline{m})]\!]$ qui ne vaut pas **undef**.

Il y a un nombre fini de tels symboles f. On utilise le principe de la preuve du théorème 5.7, en utilisant 2 rubans par tel symbole f d'arité ≥ 1 (comme pour le symbole x dans la preuve de ce théorème), un ruban par symbole d'arité 0 (comme pour le symbole x_0), en plus de rubans de travail en nombre fini.

Pour un symbole f d'arité $r \geq 1$, les deux rubans codent des mots de la forme

$$\ldots \mathbf{BB}\overline{a}_0\mathbf{B}\overline{a}_1\mathbf{B}\cdots\mathbf{B}\ldots\overline{a}_k\mathbf{BB}\cdots .$$

Le second ruban code un mot de la forme

$$\ldots \mathbf{BB}[\![(f, \overline{a}_0)]\!]\mathbf{B}[\![(f, \overline{a}_1)]\!]\mathbf{B}\cdots[\![(f, \overline{a}_k)]\!]\mathbf{BB}\cdots$$

où $\overline{a}_i$ désigne un r-uplet d'éléments de M, et $[\![(f, \overline{a}_i)]\!]$ le contenu de l'emplacement $(f, \overline{a}_i)$ (et donc désigne un élément de M).

Comme dans la preuve du théorème 5.7, tout emplacement qui n'est pas dans la liste codée par ces deux rubans correspond à une valeur indéfinie, car jamais encore accédée.

Comme dans la preuve du théorème 5.7, on simule instruction par instruction de l'algorithme, en utilisant éventuellement les rubans de travail : les mises à jour sont calculées en utilisant les anciennes valeurs dans les listes, avant d'écraser par les nouvelles valeurs en masse pour simuler les instructions **par endpar**.

Le calcul des relations et des fonctions de la structure de l'algorithme est réalisé en utilisant les relations et les fonctions de la structure par la machine de Turing. □

Remarque 5.3 *La preuve du théorème 5.7 travaille sur les codages binaires des entiers et des emplacements. Dans le théorème 5.8, on travaille directement sur des éléments de* M.

5.8 Synthèse du chapitre

En résumé, nous venons d'obtenir[3].

[3]Bien entendu, il faut comprendre derrière cette terminologie impropre que chacun de ces modèles correspond à un algorithme, et que tout algorithme peut être simulé par chacun de ces modèles.

Corollaire 5.3 *Sur une structure arbitraire* $\mathfrak{M} = (M, f_1, \cdots, f_u, r_1, \cdots, r_v)$, *les modèles suivants se simulent deux à deux*

1. *Les machines de Turing*
2. *Les machines à* $k \geq 2$ *piles*
3. *Les machines RAM*
4. *Les algorithmes au sens du chapitre 4*

Corollaire 5.4 *Sur une structure finie, ou sur un alphabet fini, les modèles suivants se simulent deux à deux*

1. *Les machines de Turing*
2. *Les machines à* $k \geq 2$ *piles*
3. *Les machines à compteurs*
4. *Les machines à* 2 *compteurs*
5. *Les machines RAM*
6. *Les algorithmes au sens du chapitre 4*

5.9 Notes bibliographiques

Le modèle de machine de Turing est dû à [Turing, 1936]. L'idée de machines de Turing travaillant sur une structure du premier ordre arbitraire est due à [Goode, 1994], et se trouve développée (moins formellement mais peut être aussi clairement) dans [Poizat, 1995]. Dans le reste du chapitre, nous avons utilisé ici diverses sources dont essentiellement [Jones, 1997] et [Hopcroft et al., 2001]. La preuve de l'équivalence entre algorithmes et machines de Turing est due et inspirée de [Dershowitz and Gurevich, 2008]. La notion d'algorithme utilisée ici est, comme nous l'avons dit dans le chapitre 4 due à [Gurevich, 2000].

Bibliographie

[Dershowitz and Gurevich, 2008] Dershowitz, N. and Gurevich, Y. (2008). A natural axiomatization of computability and proof of Church's Thesis. *The Bulletin of Symbolic Logic*, 14(3) :299–350.

[Goode, 1994] Goode, J. B. (1994). Accessible telephone directories. *The Journal of Symbolic Logic*, 59(1) :92–105.

[Gurevich, 2000] Gurevich (2000). Sequential abstract-state machines capture sequential algorithms. *ACMTCL : ACM Transactions on Computational Logic*, 1.

[Hopcroft et al., 2001] Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2001). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2nd edition.

[Jones, 1997] Jones, N. (1997). *Computability and complexity, from a programming perspective*. MIT press.

[Poizat, 1995] Poizat, B. (1995). *Les petits cailloux : Une approche modèle-théorique de l'Algorithmie*. Aléas Editeur.

[Turing, 1936] Turing, A. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2) :230–265. Reprinted in Martin Davis. The Undecidable : Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions. Raven Press, 1965.