ANALYSE D'ALGORITHMES

Johann PAVIS (selon source : D. JANIN)

Table des matières

Chapitre 1

ANALYSE D'ALGORITHMES

Enjeux sous-jacents :
→ aider à l'écriture de programmes corrects
→ développer la vision abstraite qu'on peut avoir d'un programme

Deux approches
→ Vérifier ou prouver qu'un algorithme satisfait une certaine **spécification** (description plus ou moins formelle de ce que doit faire l'algorithme)
→ Synthétiser un algorithme, le cas échéant à l'aide d'outils plus ou moins automatiques, à partir de spécification qu'il devra satisfaire (approche effectivement utilisée en base de données, compilation, contrôle/commande,...)

Pour cela, on a besoin d'un modèle d'algorithmes. On distinguera deux cas :

1. algorithmes à mémoire bornée (théorie des automates d'états finis)
2. algorithmes à mémoire non bornée (while-program)

Chapitre 2

MOTS, LANGAGES ET AUTOMATES

Étude des algorithmes à mémoire bornée (automates d'états finis)

Exemple Un passeur dispose d'une barque. Il peut transporter dans celle-ci deux personnes, **lui compris**. Il doit faire parvenir de l'autre côté de la rivière le loup, la fillette et le gâteau, en respectant les configurations interdites suivantes :

- loup + fillette sur la même rive
- fillette + gâteau sur la même rive

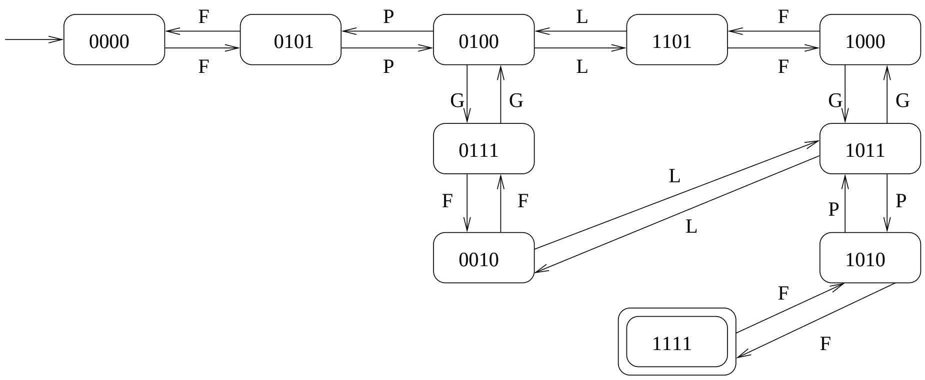
Question : Trouver tous les programmes qui permettent au passeur de faire changer de rives tout le monde.

On dispose ici :

⋆ des états possibles du système (2^4 états), codés en binaire sur 4 bits : $x_1x_2x_3x_4$

$$\begin{cases} \text{état initial : 0000} \\ \text{état final : 1111} \\ \text{états interdits : 1100,0110,1110,0001,1001,0011} \end{cases}$$

⋆ des actions possibles du système : traversée de $L, F, G + P$ ou P seul



Un programme correct pour le passeur est le suivant : $FPGGLGFLGFLPF$.

Un autre : $FPLFGPF$ (ce programme est par ailleurs **minimal**), ou encore $FPGFLPF$ (lui aussi minimal).

Tous les programmes

$\rightarrow$ les traces sur l'alphabet $\{L, F, P, G\}$ des chemins allant de l'état initial à l'état final.

Questions sous-jacentes

- quels ensembles de séquence (de mots) peut-on ainsi décrire ?
- peut-on les décrire d'autres manières ?
- est-il toujours nécessaire de distinguer deux états donnés ?
- *etc*

2.1 Mots et langages

2.2 Mots

Un **alphabet** est un ensemble fini de lettres

Un **mot** est une séquence de lettres (éventuellement vide). On note à cet effet ε le mot vide, unique séquence de lettres sans mot.

Exemple

- $A = \{a, b\} \rightarrow$ alphabet
- un mot : $u = abaa$
- un autre mot : $v = abba$
- Produit de concaténation

A partir de deux mots u et v de A^*, on définit la concaténation de u et v de la manière suivante : $w = u.v$ est le mot constitué des lettres de u suivies de celles du mot v (avec les deux mots précédents, on obtient : $u.v = abaaabba$).

Remarque Le mot vide est neutre pour la concaténation :

$$\forall u \in A^* \ , \ u.\varepsilon = \varepsilon.u = u$$

La concaténation est associative :

$$\forall (u, v, w) \in (A^*)^3 \ , \ u.(v.w) = (u.v).w$$

On dit alors que $< A^*, ., \varepsilon >$ est un **monoïde**.

2.2.1 Langages

Un **langage** est un ensemble de mots.

Exemple

$\{ab\} \subset A^*$
$\emptyset$, $\{\varepsilon\}$, $(aa)^*$
$\{u \in A^* / |u|_a = |u|_b\}$

2.2.1.1 Opérations sur les langages

Soient deux langages L_1 et L_2. On définit alors l'union et le produit de ces deux langages :

⋆ $L_1 + L_2 = L_1 \cup L_2$
⋆ $L_1.L_2 = \{u.v \in A^* / u \in L_1 \ , \ v \in L_2\}$

2.2.1.2 Étoile

On pose, pour tout langage L : $\left\{ \begin{array}{l} L^0 = \{\varepsilon\} \\ \forall n \in \mathcal{N} \ , \ L^{n+1} = L^n.L \end{array} \right.$

Ainsi, L^n est l'ensemble des mots que l'on peut construire par concaténation de n mots de L.

On définit alors :

$$L^* = \bigcup_{n \in \mathcal{N}} L^n$$

L^* est l'ensemble des mots que l'on peut construire par concaténation d'un nombre quelconque de mots de L.

Remarque La notation A^* est cohérente avec cette définition.

Dans la suite, on note (abusivement) 'a' aussi bien la lettre a, le mot d'une lettre a et le langage réduit à une lettre $\{a\}$

2.2.1.3 Expressions rationnelles

On appelle **expression rationnelle** toute expression construite à l'aide :
→ de lettres de A (langage vide inclus)
→ de somme(s), produit(s) ou étoile(s) d'expressions rationnelles plus simples

Toute expression rationnelle définit un langage. On dira qu'un langage $L \subset A^*$ est rationnel s'il peut être décrit à l'aide d'une expression rationnelle.

Exemple
$(a+b)^* \rightarrow$ langage de tous les mots construits avec des a et des b
$b^*(ab^*a)b^* \rightarrow$ langage des mots sur $\{a,b\}$ ayant un nombre pair de a
$(aa+bb)^*((ab+ba).(aa+bb)^*.(ab+ba))^*.(aa+bb)^* \rightarrow$ ensemble des mots ayant un nombre pair de a et de b

2.3 Automates d'états finis

Un automate d'états finis est un quintuplet $\mathcal{A} =< Q, q_0, A, \delta, F >$ où :
- Q est un ensemble (fini) d'états
- q_0 l'état initial
- A l'alphabet
- δ une fonction de transition ($\delta : Q \times A \rightarrow \mathcal{P}(Q)$)
- F l'ensemble des états finaux

Dans la représentation graphique, on utilise **des arcs étiquetés par des lettres** (transitions) pour relier les **sommets** (les différents états de l'automate). On distingue l'état initial par une flèche entrante et les états finaux finaux par un sur-entourage.

On dit qu'un mot u est **reconnu**, lorsqu'il existe un **chemin de trace** u allant de q_0 à un état final.

On note $L(\mathcal{A})$ l'ensemble des mots reconnus par l'automate $\mathcal{A}$. Formellement, en définissant la fonction $\delta^* : Q \times A \rightarrow \mathcal{P}(Q)$ par :

$$\begin{cases} \forall q \in Q \ , \ \delta^*(q, \varepsilon) = \{q\} \\ \forall a \in A, (q, q') \in Q^2 \ , \ \delta^*(q, a.u) = \cup \delta^*(q', u) \text{ (avec } q' \in \delta(q, a)) \end{cases}$$

On a alors : $\boxed{L(\mathcal{A}) = \{u \in A^* / \delta^*(q_0, u) \cap F \neq \emptyset}$

Théorème 1 *Soit $L \subset A^*$. Il y a lors équivalence entre :*

1. *L est **rationnel***
2. *L est **reconnaissable par automate d'états finis**, i.e. :*
   $\exists \mathcal{A} =< Q, q_0, A, delta, F >$ vérifiant :

$$L = \{u \in A^* / \delta^*(q_0, u) \cap F \neq \emptyset\} = L(\mathcal{A})$$

Preuve

(1) $\Rightarrow$ (2) : Preuve par récurrence sur la structure des expressions rationnelles.

Par hypothèse de récurrence, on suppose qu'on dispose de 2 automates $\mathcal{A}_1$ et $\mathcal{A}_2$ défini sur le même alphabet A et reconnaissant respectivement les langages L_1 et L_2.

Construisons un automate $\mathcal{A}$ qui reconnaît $L(\mathcal{A}) = L_1 + L_2$.

idée : définir l'automate $\mathcal{A}$ comme la mise en parallèle des automates $\mathcal{A}_1$ et $\mathcal{A}_2$ (*i.e.* faire tourner ces deux automates en même temps).

Pour cela, on pose :
- $Q = Q_1 \times Q_2$
- $q_0 = (q_{0,1}, q_{0,2})$
- $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$
- $\forall (q_1, q_2) \in Q^2, \forall a \in A, \delta((q_1, q_2), a) = \delta_1(q_1, a) \times \delta_2(q_2, a)$

Alors, on a bien : $L(\mathcal{A}) = L_1 + L_2 = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$

En effet, pour tout mot $u \in A^*$, l'ensemble $\delta^*(q_0, u)$ des états de $\mathcal{A}$ que l'on peut atteindre en lisant u est, par construction : $\delta_1^*(q_{0,1}, u) \times \delta_2^*(q_{0,2}, u)$.

Avec la définition de F, on a :

$$\delta^*(q_0, u) \cap F \neq \emptyset \Leftrightarrow \begin{cases} \delta_1^*(q_{0,1}, u) \cap F \neq \emptyset \\ \text{ou} \\ \delta_2^*(q_{0,2}, u) \cap F \neq \emptyset \end{cases} \Leftrightarrow ((u \in L(\mathcal{A}_1)) \ (u \in L(\mathcal{A}_2)))$$

Construisons à présent l'automate reconnaissant $L = L_1.L_2$, où L_1 et L_2 sont respectivement reconnus par les automates $\mathcal{A}_1$ et $\mathcal{A}_2$.

On définit l'automate $\mathcal{A}$ reconnaissant L comme la mise en séquence des automates $\mathcal{A}_1$ et $\mathcal{A}_2$. On pose à cet effet :

- $Q = Q_1 \cup Q_2$ (ou $Q_1 \cup Q_2$ si $Q_1 \cup Q_2 = \emptyset$)
- $q_o = q_{0,1}$
- $F = \begin{cases} F_2 \text{ , si } \varepsilon \in L_2 \\ F_1 \cup F_2 \text{ , sinon} \end{cases}$
- $\forall q \in Q_1, \forall a \in A, \delta(q,a) = \begin{cases} \delta_1(q,a) \text{ , si } q \notin F_1 \\ \delta_1(q,a) \cup \delta_2(q_{0,2},a) \text{ , sinon} \end{cases}$
- $\forall q \in Q_2, \forall a \in A, \delta(q,a) = \delta_2(q,a)$

Avec cette définition, pour tout mot $u \in A^*$:

$$\delta^*(q_0,u) = \delta_1^*(q_{0,1},a) \bigcup \left(\bigcup_{\substack{v.w=u \\ \delta^*(q_{0,1},w) \cap F_1 \neq \emptyset}} \delta^*(q_{0,2},w) \right)$$

Ainsi, on a : $u \in L(\mathcal{A}) \Leftrightarrow u \in (L(\mathcal{A}_1).L(\mathcal{A}_2))$

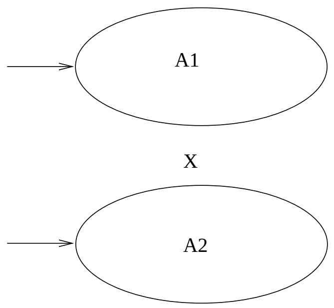
Construisons maintenant l'automate reconnaissant l'étoile d'un langage L, où L est reconnu par l'automate $\mathcal{A}_1$.

On pose à cet effet :

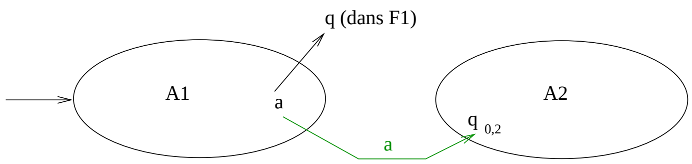
- $Q = \{q_0\} \cup Q_1$ (où $q_0 \notin Q_1$)
- $F = \{q_0\}$
- $\forall a \in A, \delta(q_0,a) = \delta_1(q_{0,1},a)$
- $\forall q \in Q_1, \delta(q,a) = \begin{cases} \delta_1(q,a) \text{ , si } q \notin F \\ \delta_1(q,a) \cup \{q_0\} \text{ , sinon} \end{cases}$

On vérifie alors que $L(\mathcal{A}) = (L(\mathcal{A}))^*$.

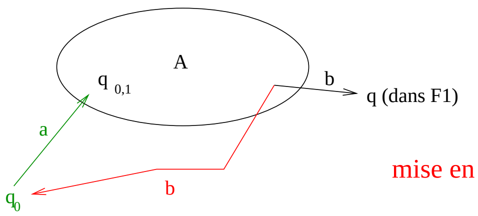
Pour résumer



produit parrallèle par l'union



produit séquentiel par le produit



mise en boucle par l'étoile

Réciproquement

(2) $\Rightarrow$ (1)

Soit l'automate $\mathcal{A} =< Q, q_0, Q, \delta, F >$. On veut construire une expression rationnelle qui définit le langage reconnu par $\mathcal{A}$, *i.e.* le langage $L(\mathcal{A})$.

Une méthode (du type résolution de système d'équations / pivot de Gauss) pour passer d'un automate à une expression rationnelle équivalente est la suivante :

- pour tout état $q \in Q$, posons (comme inconnue) dans $\mathcal{P}(A^*)$ L_q, langage des des mots reconnus par $\mathcal{A}$ à partir de l'état q :

$$L_q = \{u \in A^*/\delta^*(q,a)\ capF \neq \emptyset\}$$

- **<u>idée</u>** : décrire la fonction de transition et les états finaux à l'aide d'équations sur les langage L_q
  1. $q \in F \Leftrightarrow \varepsilon L_q$
  2. $(q' \in \delta(q,a)) \Rightarrow a.L'_q \subseteq L_q$
- Pour tout état $q \in Q$, on q donc :

$$L_q = \begin{cases} \varepsilon + \sum_{a \in A} \sum_{q' \in \delta(q,a)} a.L'_q \text{ , si } q \in F \\ \sum_{a \in A} \sum_{q' \in \delta(q,a)} a.L'_q \text{ , sinon} \end{cases}$$

  C'est un système à n inconnues (n est le nombre d'états de l'automate) ensemblistes. En général, ce système d'équations admet plusieurs solutions.

  On admet que la solution qui nous intéresse peut-être obtenue à partir de la règle suivante :

$$X = L.X + M \Rightarrow X = L^*.M$$

  Ainsi, de proche en proche (avec le cas échéant l'utilisation de la distributivité de '.' sur '+'), on résout le système d'équations : $L_{q_0} = L(\mathcal{A})$.

2.4 Résidus d'un langage, automate minimal

Question Existe-t-il une caractérisation plus directe des langages rationnels / reconnaissables ?

La réponse est oui, comme nous allons le voir par la suite

<u>Définition</u> 2 *Jetant donnés $L \subseteq A^*$ et $u \in A^*$, on définit le résidu de L par u comme le langage noté $u^{-1}.L$:*

$$u^{-1}.L = \{v \in A^*/u.v \in L\}$$

<u>Théorème</u> 3 *Un langage $L \subseteq A^*$ est rationnel / reconnaissable ssi $RES(L) = \{u^{-1}.L \subseteq A^*/u \in A^*\}$ est de cardinal fini.*

Preuve

$\Leftarrow$ Supposons que $RES(L)$ est fini.

Remarquons tout d'abord que : $u \in L \Leftrightarrow \varepsilon \in u^{-1}.L$

<u>idée</u> : définir l'automate des résidus de L dont les états sont les résidus de L : $\mathcal{A}_L =< Q, q_0, A, \delta, F >$, avec :

- $Q = RES(L)$
- $q_0 = L$
- $\forall M \in RES(L), \forall a \in A, \delta(M, a) = \{a^{-1}.M\}$
  (*Remarque* : pour certain $u \in A^*$, on a $M = u^{-1}.L$; donc $\delta(M, a) = \{a^{-1}.(u^{-1}.L)\} = \{(ua)^{-1}.L\}$. C'est donc bien un résidu)

  Question Que dire alors de $\delta^*(q_0, u)$ pour $u \in A^*$?
  On constate (preuve par récurrence sur la longueur de u) que $\delta^*(q_0, u) = \{u^{-1}.L\}$
  En effet :
  $\rightarrow$ $\delta^*(q_0, \varepsilon) = \{q_0\} = \{L\} = \{\varepsilon^{-1}.L\}$
  $\rightarrow$ si $\delta^*(q_0, u) = \{u^{-1}.L\}$
  $\forall a \in A, \delta^*(q_0, u.a) = \delta(\delta^*(q_0, u), a) = \{\delta(u^{-1}.L, a)\} = \{(ua)^{-1}.L\}$

- $F = \{u^{-1}.L / u \in A^*, \varepsilon \in u^{-1}.L\}$
  On a alors :

$$\forall u \in A^*, \delta^*(q_0, u) \cap F \neq \emptyset \Leftrightarrow \varepsilon \in u^{-1}.L \Leftrightarrow u \in L$$

Ainsi : $L(\mathcal{A}_L) = L$

Réciproquement

$\Rightarrow$ Supposons que L est rationnel. Soit alors $\mathcal{A} =< Q, q_0, A, \delta, F >$ un automate d'états finis reconnaissant L.

Soit $u \in A^*$. Que dire de $u^{-1}.L$?

$$u^{-1}.L = \{v \in A^* / \delta^*(\delta^*(q_0, u), v) \cap F \neq \emptyset\} = \bigcup_{q \in \delta^*(q_0, u)} L_q$$

$RES(L)$ est donc fini et, de plus, $Card(RESL(L)) \leq 2^{Card(Q)}$

Remarques

- Tout automate reconnaissant L aura au minimum
  $|log_2 \left[Card\left(RES(L)\right)\right]|$ états.

- A partie de $\mathcal{A}$ déterministe (*i.e.* pour tout état q et toute lettre 'a', $|\delta(q,a)| = 1$), la même construction donne :

$$\forall u \in A^*, u^{-1}.L = L_q$$

où q est l'unique état vérifiant : $\delta^*(q_0, u) = \{q\}$

Donc : $\boxed{Card(Q) \geq Card(RES(L))}$.

Corollaire 4 *Pour tout langage $L \subseteq A^*$, il existe un automate déterministe qui redonnait L (c'est l'automate des résidus).*

De plus, le nombre d'états de l'automate des résidus est minimum : il n'existe pas d'automates déterministes reconnaissant L avec moins d'états.

Questions induites A partir de $\mathcal{A}$ automate reconnaissant un langage donné L,

1. comment construire un automate déterministe $\mathcal{A}_d$ tel que $L(\mathcal{A}_d) = L(\mathcal{A})$?
2. en supposant $\mathcal{A}$ déterministe, comment fabriquer l'automate minimal reconnaissant $L(\mathcal{A})$

2.5 Automates déterministes *vs* automates non déterministes

On a vu sur un exemple une transformation d'automate non déterministe en automate déterministe.

Théorème 5 *Tout automate non déterministe $\mathcal{A} =< Q, q_0, A, \delta, F >$ est équivalent à un automate déterministe $\mathcal{A}^d =< Q^d, q_o^d, A, \delta^d, F^d >$ tel que : $L(\mathcal{A}) = L(\mathcal{A}^d)$*

De plus, on a : $|Q^d| \leq 2^{|Q|}$

Preuve

Soit donc $\mathcal{A} =< Q, q_0, A, \delta, F >$ un automate donné avec un nombre fini d'états.

On définit l'automate $\mathcal{A}^d$ de la façon suivante :

* $Q^d = \mathcal{P}(Q)$

* $q_0^d = \{q_0\}$
* $\forall a \in A, \forall P \subseteq Q, \delta^d(P, A) = \bigcup_{q \in P} \{\delta(q, a)\}$
* $F^d = \{P \subseteq Q / F \cap P \neq \emptyset\}$

De façon analogie au cas non déterministe, on définit la fonction δ^{d*} :

$$\delta^{d*} : Q^d \times A^* \to P \subseteq Q$$

$$\forall a \in A, u \in A^*, P \subseteq Q, \delta^{d*}(P, a.u) = \delta^{d*}(\delta^d(P, a), u)$$

On remarque que :

$$\forall u \in A^*, \delta^*(q_0, u) = \delta^{d*}(\{q_0\}, u)$$

Ainsi, vu la définition de F^d, un mot est reconnu par le premier automate ssi il est reconnu par le second : $\forall u \in A^*, u \in L(\mathcal{A}) \Leftrightarrow u \in L(\mathcal{A}^d)$

Un automate déterministe reconnaissant $L_n = (a+b)^*.a.(a+b)^{n-1}$ possède au moins 2^n états. Il y en a un avec exactement 2^n états.

Éléments complémentaires de comparaison des automates déterministes et non déterministes

Étant donnés un automate $\mathcal{A}$ et un mot $u \in A$, quelle est la complexité (en temps, en espace) pour décider : $u \in L(\mathcal{A})$?

1. Stockage de l'automate
   - cas non déterministe : $\leq |A| \times |Q|^2$

$$\sum_{(q,a) \in Q \times A} |\delta(q, a)| + 1$$

   - cas déterministe : $\leq |Q| \times |A| \times \log |Q|$
2. Complexité en temps
   - cas déterministe : $|u| \to$ lecture de lettres
   - cas non déterministe : par l'algorithme de TARJAN $\to \leq |u| \times |Q|^2$

2.6 Limites du pouvoir d'expression des machines à états finaux

Exercice Montrer que le langage des mots de la forme $\{a^n b^n / n \in \mathbb{N}\}$ n'est pas reconnaissable par automate d'états finis.

Preuve (par l'absurde)

Supposons ce langage reconnaissable (on le notera L par la suite).

Soit $\mathcal{A}$ un automate d'états finis reconnaissant L. On note k le nombre d'états de cet automate.

Soit le mot : $w = a^k b^k \in L$. Donc w est reconnu par $\mathcal{A}$.

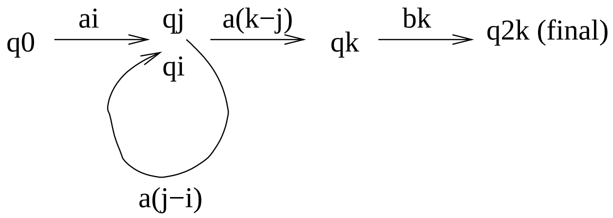
Soit $q_0 q_1 \ldots q_2 k$ la séquence des états rencontrés en lisant ce mot w dans $\mathcal{A}$.

On a donc : $\forall i \in [0, 2k], q_i = \begin{cases} \delta^{(}q_0, a^i) \text{ , si } k \geq i \\ \delta^*(q_0.a^k b^{k-i}) \text{ , si } k+1 \leq i \leq k \end{cases}$

On sait également : $w \in L(\mathcal{A}) \Rightarrow q_{2k} \in F$

On sait $|Q| = k$. Donc :

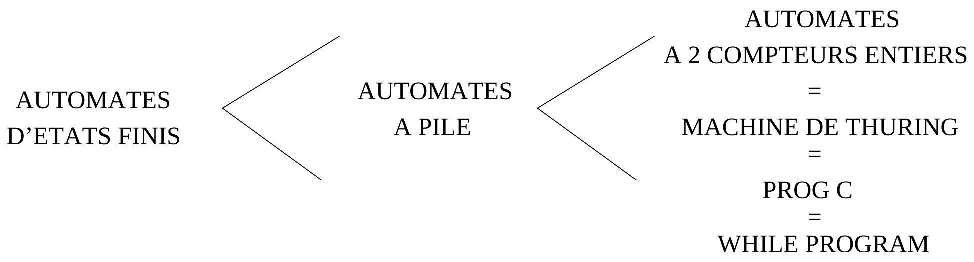
$$\exists (i,j) \in [0,k]^2, i \leq j, q_i = q_j$$



On constate donc que $a^{k-(j-i)b^k}$ est un mot reconnu par l'automate $\mathcal{A}$. Ceci est **ABSURDE**, ce mot n'appartenant pas au langage reconnu par $\mathcal{A}$.

Remarque Dans la littérature, on parle du lemme de l'étoile (ou lemme de pompage)

Autres modèles plus riches



<u>**Thèse de CHURCH**</u> **6** *Les fonctions ($\simeq$ les algorithmes) calculables sont exactement les fonctions implémentables à l'aide de machines de Thüring (ou de programmes C).*

Corollaire 7 *Certaines fonctions ne sont pas calculables.*

Il y a une quantité **dénombrable** de programmes C, alors que l'ensemble des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ **n'est pas** dénombrable.

En effet, supposons qu'on dispose d'une énumération des fonctions de $\mathbb{N}$ dans $\mathbb{N}$: $(f_n)_{n\in\mathbb{N}}$.

On pose alors : $g : \mathbb{N} \to \mathbb{N}$ telle que $\forall n \in \mathbb{N}, g(n) = f_n(n) + 1$

On constate alors que : $\forall n \in \mathbb{N}, g \neq f_n$. Ceci est donc en contradiction avec l'hypothèse d'énumération.

Chapitre 3

LES WHILE-PROGRAMS

3.1 Syntaxe

Soit Var un ensemble de symboles de variables (supposées entières). Soit $Expr$ l'ensemble des expressions entières qu'on peut construire avec des variables de Var, des constantes entières et des opérateurs $(+, -, *, /n \leq, \geq, ==, etc)$

On définit alors l'ensemble des while-programs de la façon suivante :
- `nop` (**no** **op**eration)
- $x := e$ (**affectation**)

(*while-programs élémentaires*)

Étant donnés p et p' deux while-programs déjà construits et e une expression :
- $p; p'$ (**composition séquentielle**)
- *if e then p else p'* (**conditionnelle**)
- *while e do p* (**boucle while**)

(*constructeurs de while-programs*)

3.2 Sémantique

On appelle **environnement** une fonction σ telle que : $\sigma : Var \rightarrow \mathbb{Z}$
On note $\perp$ l'environnement non défini.

En posant $Env = (Var \rightarrow \mathbb{Z}) \bigcup \{\perp\}$, on définit la sémantique de tout while-program p comme une fonction $Sp : Env \rightarrow Env$ et vérifiant :
$Sp(\perp) = \perp$

La fonction $Sp : Env \rightarrow Env$ est définie par induction sur la structure de p :

$$\forall \sigma \in \mathcal{F}(Var \rightarrow Env), \begin{cases} S_{nop}(\sigma) = \sigma \\ S_{x:=e}(\sigma) = \sigma[x \rightsquigarrow e] \\ S_{p_1;p_2}(\sigma) = S_{p2} \circ S_{p1}(\sigma) \\ S_{if\ e\ then\ p_1\ else\ p_2}(\sigma) = \begin{cases} S_{p_1}(\sigma) \text{ , si } \sigma(e) \neq 0 \\ S_{p_2}(\sigma) \text{ , sinon} \end{cases} \\ S_{while\ e\ do\ p}(\sigma) = \lim_{n \rightarrow \infty} S_p^n(\sigma) \end{cases}$$

avec : $\sigma[x \rightsquigarrow e](y) = \begin{cases} \sigma(y) \text{ , si } x \not\cong y \\ \sigma(e) \text{ , sinon} \end{cases}$

et $\begin{cases} S_p^0(\sigma) = \perp \\ S_p^{n+1}(\sigma) = \begin{cases} \sigma \text{ , si } \sigma(e) \neq 0 \\ S_p^{n+1}(S_p(\sigma)) \text{ , sinon} \end{cases} \end{cases}$

On notera que la limite existe bien, de par le lemme suivant :

Lemme 8 *Pour tout environnement σ on a :*

- $\forall n \in \mathbb{N}, S_p^n(\sigma) = \perp$

ou

- $\exists n_0 \in \mathbb{N}, \forall n \geq n_0, S_p^n(\sigma) = S_p^{n_0}(\sigma)$

Chapitre 4

CORRECTION DES WHILE-PROGRAMS

4.1 Généralités

Soit p un while-program.

Question Le programme p est-il correct ?

Deux sous-problèmes
→ terminaison : le programme p s'arrête-t-il ?
→ correction partielle : s'il s'arrête, calcule-t-il ce que l'on attend ?

Idée Étudier indépendamment les deux cas : terminaison et correction partielle.

4.1.1 Terminaison

Soit $S_p : Env \to Env$ la sémantique de p. Soit $Env' \subseteq Env$ un ensemble d'environnements par ailleurs.

On doit s'assurer que $S_p(Env') \notin \perp$

Remarque Env' sera en général dans les **spécifications** du programme p.

Outils pour étudier la terminaison

$\rightarrow$ Aucun outil automatique, *i.e.* le problème de la terminaison est **indécidable**. Il n'existe pas de fonction calculable prenant en entrée des (codes de) while-programs qui, terminant, produit 1 ou 0 selon que le programme d'entrée termine ou ne termine pas sur un environnement donné ($\sigma \in Env$ tel que : $\forall x \in Var, \sigma(x) = 0$).

$\rightarrow$ Un seul outil conceptuel pour l'étude de la terminaison : le **variant**

Définition 9 *Un variant est une mesure de l'état d'un programme à un instant donné tel que :*

- $\star$ *l'ensemble des valeurs possibles de cette mesure est ordonné pour formé un ensemble bien fondé (ne comportant pas de suite infinie strictement décroissante)*
- $\star$ *l'assurance que, à chaque tour de boucle, le variant décroît strictement (à chaque appel pour les fonctions récursives)*

4.1.2 Correction partielle

"Si le programme termine et calcule bien ce ce que l'on attend"

Spécification de correction partielle

Soit $\mathcal{P}$ un ensemble tel que : $\mathcal{P} \subseteq (Env \rightarrow Env)$.

Idée $\mathcal{P}$ est l'ensemble des fonctions possiblement implémentables

Correction partielle de p

Trouver $F \in \mathcal{P}$ tel que :

$$\forall \sigma \text{ vérifiant } S_p(\sigma) \neq \perp, S_p(\sigma) = F(\sigma)$$

Remarque On peut aussi se restreindre à des environnements de départ Env' spécifié par ailleurs.

Outils disponibles

$\rightarrow$ Aucun automatique : la correction partielle est **indécidable**

$\rightarrow$ Raisonner sur les while-programs (calcul des assertions)

4.2 Calcul des assertions (Dijkstone, Hoare)

Exemple Une assertion

$$\{\varphi\}p\{\Psi\}$$

4.2.1 Assertions valides

On dit que l'assertion $\{\varphi\}p\{\Psi\}$ est **valide** lorsque pour tout environnement $\sigma : Var \to \mathbb{Z}$, si $\sigma \vDash$ (satisfait à) φ et si $S_p(\sigma) \neq \bot$, alors : $S_p(\sigma) \vDash \Psi$

4.2.2 Raisonner sur les assertions ?

Produire des assertions valides complexes à partir d'assertions valides plus simples en utilisant des règles de compositions (règles de raisonnement) validées une fois pour toute (Leibniz 1700, puis Hilbert 1900).

Idée Raisonner sur les programmes en s'appuyant sur leur structure syntaxique.

$$(\texttt{nop})\ \frac{}{\{\varphi\}\texttt{nop}\{\Psi\}} \quad \begin{array}{l} \leftarrow \textit{sans aucune hypothèse particulière} \\ \leftarrow \textit{assertion valide} \end{array}$$

$$(\ ;\ )\ \frac{\{\varphi\}p_1\{\varphi_1\}\{\Psi_1\}p_2\{\Psi\}\ \varphi_1 \Rightarrow \Psi}{\{\varphi\}p_1; p_2\{\Psi\}}$$

$$(\ \texttt{:=})\ \frac{\varphi \Rightarrow \Psi[x/e]}{\{\varphi\}x := e\{\Psi\}}$$

où $\Psi[x/e]$ désigne la propriété Ψ dans laquelle on a systématiquement remplacé x par e, sa valeur

$$(\texttt{if then else})\ \frac{\{\varphi \text{ et } e \neq 0\}p_1\{\Psi\}, \{\varphi \text{ et } e = 0\}p_2\{\Psi\}}{\{\varphi\} \text{ if } e \text{ then } p_1 \text{ else } p_2\ \{\Psi\}}$$

$$(\texttt{while})\ \frac{(\varphi \Rightarrow I)\ \{I \text{ et } e \neq 0\}p\{I\}, \{I \text{ et } e = 0\} \Rightarrow \Psi}{\{\varphi\} \text{ while } e \text{ do } p\ \{\Psi\}}$$

(où I est une formule à déterminer, appelée **invariant de boucle**)

4.2.3 Face à ce système de démonstration, trois questions

1. **Une assertion prouvable est-elle valide**

   Oui. Il suffit de vérifier (induction sur la taille de preuves) que les règles proposées sont correctes.

2. **Une assertion valide est-elle prouvable ?** (complétude, *cf* Godel pour la logique du premier ordre)

   Non. Cependant, on peut obtenir une complétude relative. Si le langage logique est suffisamment expressif, on peut se ramener, via les règles de calcul proposées, à des formules logiques.

   Idée : exprimer, en logique, pour tout programme p, toute post-condition Ψ, $w_p(p, \Psi)$ la **pré-condition la plus faible** garantissant que, en exécutant p, l'on satisfera à Ψ si on termine.

   Remarque
   - $w_p(x := e, \Psi) \equiv \Psi[x/e]$
   - $w_p(p_1; p_2, \Psi) \equiv w_p(p_1, w_p(p_2, \Psi))$
   - $w_p(\texttt{nop}, \Psi) \equiv \Psi$
   - $w_p(\textit{if } e \textit{ then } p_1 \textit{ else } p_2) \equiv (e \Rightarrow w_p(p_1, \Psi)) \vee (\neg e \Rightarrow w_p(p_2, \Psi))$
   - $w_p(\textit{while } e \textit{ do } p) \rightarrow$ pas de formule générale !!

3. **Existe-t-il une machine permettant de fabriquer automatiquement les preuves des assertions prouvables ?** (décidabilité)

   Non. En effet, l'assertion : $\{true\}p; x = 0\{x = 1\}$ n'est vraie que si le programme p ne termine pas.

Théorème 10 *La terminaison des while-programs est **indécidable**.*

Preuve

1. Pour chaque programme p, notons n_p l'entier obtenu en lisant p comme un entier binaire
2. Supposons qu'on dispose d'un programme t (pour *terminaison*) tel que :
   - t termine toujours
   - $\forall \sigma, \forall p \text{ (programme)}, \sigma(x) = n_p, S_t(\sigma)(y) = \begin{cases} 1 \text{ , si } S_p(\sigma) \neq \perp \\ 0 \text{ , sinon} \end{cases}$

3. Soit c (pour *croche patte*) le programme suivant :

$$c \equiv t; while y = 1 do nop$$

4. Soit σ un environnement tel que : $\sigma(x) = n_c$
   Calculons $\S_t(\sigma)(y)$

   $\underline{S_t(\sigma)(y) = 1}$ dans ce cas, par définition de t, on a : $S_c(\sigma) \neq \perp$.
   Donc $S_t(\sigma)(y) \neq 1 \Longrightarrow$ CONTRADICTION

   $\underline{S_t(\sigma)(y) = 0}$ dans ce cas, par définition de t, on a : $S_c(\sigma) = \perp$.
   Donc, $S_t(\sigma)(y) = 1$, car t termine toujours

Ainsi, on a démontré, par l'absurde, qu'un tel programme t de terminaison n'existe pas.