Calculabilité avancée
Théorèmes de récursion

Kévin PERROT – Aix Marseille Université – hiver 2021-22

Table des matières

1 Théorèmes de récursion

Sources : [2] (chapitre 6.1) et [1].

Pour cette section nous aurons besoin de nous souvenir des points suivants.

Il existe une *énumération des fonctions calculables*. Comme il est d'usage dans la littérature, nous noterons ϕ_e la $e^{\text{ième}}$ fonction calculable (nous avons vu qu'il existe une énumération des MT, ce qui revient au même que d'énumérer les fonctions calculables, puisque les fonctions calculables sont calculées par ces mêmes MT). Le nombre e peut être vu comme la représentation du code d'un programme. On utilisera l'égalité $\phi_e = \phi_{e'}$ lorsque les machines calculent la même fonction.

Il existe un *interpréteur* (une machine de Turing universelle) $\phi_u(n, \vec{x}) = \phi_n(\vec{x})$, qui prend en entrée la description d'une fonction calculable (le code d'une MT) et reproduit cette fonction (simule cette MT) sur le reste de l'entrée ($\vec{x}$).

Théorème s-m-n : il existe une fonction calculable $s : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ telle que $\phi_{s(m,n)}(\vec{x}) = \phi_m(n, \vec{x})$. C'est-à-dire, on peut bêtement [1] coder en dur les arguments.

1.1 Théorème du point fixe de Kleene

Théorème 1. *Pour toute fonction (totale) calculable h, il existe un programme n tel que*

$$\phi_n(\vec{x}) = \phi_{h(n)}(\vec{x}).$$

Le théorème du point fixe de Kleene nous dit que pour toute transformation algorithmique h sur les programmes, il existe un programme qui fait la même chose que son transformé. Et comme dit au début de la phrase, c'est vrai pour toute transformation h !

1. la fonction s est calculable, on suit son algorithme pour mettre en dur n dans le code de m.

Démonstration. Soit le programme m suivant : sur l'entrée $t, \vec{x}$, calculer $s(t,t)$, puis calculer $h(s(t,t))$, puis simuler la machine $h(s(t,t))$ sur l'entrée $\vec{x}$. C'est-à-dire,

$$\phi_m(t, \vec{x}) = \phi_u(h(s(t,t)), \vec{x}.$$

Alors $n = s(m,m)$ est notre point fixe, car (définition de s, ci-dessus, définition de ϕ_u)

$$\phi_{s(m,m)}(\vec{x}) = \phi_m(m, \vec{x}) = \phi_u(h(s(m,m)), \vec{x}) = \phi_{h(s(m,m))}(\vec{x}).$$

□

1.2 Quine (*self-replicating programs*)

Le théorème du point fixe de Kleene engendre un corollaire divertissant.

Définition 2. *Un* **quine** *est un programme qui affiche à l'écran son propre code source.*

Théorème 3. *Tout langage de programmation acceptable*[2] *admet des quines.*

Démonstration. Pour un programme t, considérons le programme $h(t)$ qui affiche à l'écran le code de t. La fonction h est calculable[3]. Appliquons le théorème 1 à la fonction h : il existe un programme n tel que les programmes $h(n)$ et n sont identiques, donc n affiche à l'écran le code de n. □

Les *virus informatique* sont des programmes basés sur l'auto-réplication. Dans le but de réaliser cette tâche, un virus peut contenir la construction d'un quine pour reproduire son propre code. Comment construire un tel programme en pratique ? Relisez les preuves, elles sont **constructives**, c'est-à-dire qu'elles prouvent l'existence de programmes possédant certaines propriétés tout en expliquant comment les construire. Par contre il y a besoin d'une machine universelle...

Voici comment construire un quine, composé de deux machines de Turing.
- La machine A écrit le code de la machine B sur le ruban.
- La machine B :
  1. lit son entrée w sur le ruban,
  2. calcule le code de la machine W qui écrit le mot w sur le ruban,
  3. écrit sur le ruban le code la machine W composée avec la machine w (*càd* la machine W dont l'état final est remplacé par l'état initial de w).

La définition de A dépend de celle de B, mais celle de B ne dépend pas de celle de A. On peut donc construire la machine B et obtenir son code, ce qui nous permet de constuire A qui écrit ce code. La composition des machines A et B, appelons la C, est un quine. En effet, quand on lance la machine C :
- la machine A écrit le code de B sur le ruban, atteint son état final,
- qui est l'état initial de la machine B (C étant la compsition des machines A et B), qui s'exécute donc sur l'entrée que A vient de laisser sur le ruban :
  1. lit le code de la machine B,

2. un langage de programmation est *acceptable* s'il vérifie les points précisés en début de section.

3. on peut écrire un programme h qui prend en entrée le code de t, et produit le code d'une machine qui affiche le code de t.

2. calcule le code de la machine qui écrit le code de B sur le ruban, qui se trouve être la machine A,
3. écrit sur le ruban la composition des machines A et B, qui se trouve être C.

Lui-même.

Un jeu d'initié[4] consiste à trouver le plus petit quine dans son langage préféré... ci-dessous quelques exemples de quine en C, Bash, Haskell, Java, OCaml, Python et en français (des sauts de lignes ont été ajoutés).

```
#include<stdio.h>
main(){char*a="#include<stdio.h>%cmain(){char*a=%c%s%c;printf(a,10,34,a,34);
}";printf(a,10,34,a,34);}

z=\' a='z=\\$z a=$z$a$z\; eval echo \$a'; eval echo $a

s="main=putStr (['s', '='] ++ show s ++ [';'] ++ s)";
main=putStr (['s', '='] ++ show s ++ [';'] ++ s)

class Quine{public static void main(String[] args){char n=10;char b='"';
String a="class Quine{public static void main(String[] args)
{char n=10;char b='%c';String a=%c%s%c;System.out.format(a,b,b,a,b,n);}
}%c";System.out.format(a,b,b,a,b,n);}}

(fun s -> Printf.printf "%s %S;;" s s)
 "(fun s -> Printf.printf \"%s %S;;\" s s)";;

a='a=%r;print(a%%a)';print(a%a)

Recopier puis recopier entre guillemets la phrase
« Recopier puis recopier entre guillemets la phrase »
```

1.3 Avoir accès à son propre code

On peut appliquer le théorème 1 pour montrer qu'un programme peut avoir accès à son propre code, dans le sens où il suffit de supposer qu'il prend en premier argument son propre code, et alors le théorème 1 nous construit un programme équivalent qui se passe du premier argument (ouf).

Théorème 4. *Pour tout m, il existe n tel que $\phi_n(\vec{x}) = \phi_m(n, \vec{x})$.*

Utilisation : on construit m dont on suppose qu'elle prend en premier argument son propre code, le théorème 4 nous donne n qui fait la même chose que m, avec vraiment son propre code à elle intégré à la place du premier argument (« *filled in automagically* »).

Démonstration. Soit un m arbitraire, on considère la fonction calculable $h(t) = s(m, t)$. Le théorème 1 nous donne n tel que $\phi_n(\vec{x}) = \phi_{h(n)}(\vec{x}) = \phi_{s(m,n)}(\vec{x}) = \phi_m(n, \vec{x})$. □

Ainsi on peut considérer qu'une machine peut faire appel à l'instruction

4. essayez d'écrire un quine, vous verrez que ce n'est pas facile !

« obtenir mon propre code »

puisqu'en applicant le théorème 4 on obtient un programme équivalent qui a effectivement accès à son propre code. Cette instruction est très puissante pour construire des preuves d'indécidabilité : informellement, pour tout programme f qui *soi-disant* décide une propriété d'un programme qui lui est donné en entrée, on peut construire un programme g « pathologique » qui :

1. obtient son propre code g,
2. calcule le résultat de f pour l'entrée g,
3. fait le contraire.

Alors f se trompe sur l'entrée g, donc un tel programme f n'existe pas (démonstration par l'absurde en supposant l'existence de f).

Théorème 5. *L'arrêt des programmes n'est pas décidable.*

Démonstration. Par l'absurde, si on suppose que le programme h décide, étant donné un programme f et une entrée $\vec{x}$, si le calcule de f sur $\vec{x}$ s'arrête, alors on peut construire le programme g qui, sur une entrée $\vec{x}$:

1. obtient son propre code g,
2. calcule $\phi_h(g, \vec{x})$ (on a supposé que h est calculable),
3. si h prédit l'arrêt, alors g entre dans une boucle infinie, sinon g s'arrête.

Alors pour tout $\vec{x}$ le résultat de $\phi_h(g, \vec{x})$ est incorrect, une contradiction. □

Définition 6. *Pour une machine M, la* taille *de la description de M est le nombre de lettres du mot $\langle M \rangle$ (ou bien, pour une énumération $(\phi_e)_{e\in\mathbb{N}}$ des MT, la taille d'une description de ϕ_e est simplement le nombre e). Une machine M est* minimale *s'il n'existe pas de machine de plus petite taille équivalente*[5] *à M. Soit*

$$MIN = \{\langle M \rangle \mid M \textit{ est une MT minimale}\}.$$

Attention, cette définition dépend de notre encodage $\langle M \rangle$ ou de notre énumération $(\phi_e)_{e\in\mathbb{N}}$.

Théorème 7. *MIN n'est pas récursivement énumérable.*

Démonstration. Par l'absurde, si on suppose que le programme m énumère MIN, alors on peut construire le programme f qui, sur l'entrée x :

1. obtient son propre code f,
2. lance la machine m qui énumère MIN, et attend de la voir énumérer le code d'une machine g plus grande que son propre code f,
3. simule g sur l'entrée x.

Puisque l'ensemble MIN est infini, il énumère nécessairement le code de machines de tailles arbitraires, et le second point termine. On a f qui est équivalente à g mais a un code plus petit, ce qui est en contradiction avec le fait que m ait énuméré g. □

Références

[1] D. Madore. The fixed-point theorem. `http://www.madore.org/~david/computers/quine.html#sec_fp`, (consulté en février 2019).

[2] M. Sipser. *Introduction to the theory of computation.* Course Technology, 2006.

5. c'est-à-dire qui reconnaît le même langage ou qui calcule la même fonction.