Théorème de récursion de Kleene, preuve et applications

Paul Patault

8 mars 2022

Résumé

Il s'agit d'étudier le théorème de récursion de Kleene, l'une des premières formalisations annonçant le droit de l'usage de la récursion dans les langages de programmation.

1 Introduction

Un *quine* est un programme qui, exécuté sans entrée, produit *uniquement* un affichage *complet* de son propre code source. Lors de son discours pour sa remise de prix Turing, Ken Thompson rappelle leur importance au travers de leur possible emploi inquiétant dans le domaine de la sécurité informatique [Tho84]. Les *quines* sont en fait une application directe du théorème de récursion de Kleene que nous allons maintenant présenter.

1.1 Définitions

Nous utiliserons tout au long de ce document les notations définies telles que :

Notation 1. *φ_i est la machine de Turing « indice i », c'est à dire que si l'on énumère de façon arbitraire les machines de Turing, il faudrait s'arrêter à la i-ème. Ainsi, si l'on sait que l'on cherche une machine de Turing, la valeur de i est une suffisante pour y accéder.*

Notation 2. *$\varphi_x(y)$ est la valeur calculée par la machine de Turing φ_x avec y pour entrée.*

Notation 3. *$\langle x, y\rangle$ correspond à une paire d'entier. Cette pair est donc une suffisante pour pouvoir accéder à la valeur de la machine de Turing φ_x s'exécutant sur l'entrée y.*

2 Reculer pour mieux sauter

Un court détour par ce lieu familier nous permettra de mieux situer le problème qui est posé. Jetons alors un œil, à notre bon vieux théorème.

Théorème 1. *Le problème de l'arrêt est indécidable.*

Démonstration. Notons $\ulcorner \varphi_x(y) \urcorner$ comme codage de $\langle x, y \rangle$, et utilisons cette notation pour prouver une nouvelle fois l'indécidabilité du problème de l'arrêt. La question est alors, existe-t'il une fonction calculable h tel que pour tout x, et pour tout y,

$$h(\ulcorner \varphi_x(y) \urcorner) = \begin{cases} 1 & \text{si } \varphi_x(y)\downarrow \\ 0 & \text{sinon.} \end{cases}$$

En supposons par l'absurde que cette fonction existe, nous pouvons définir la fonction semi-calculable g telle que :

$$g(x) = \begin{cases} 0 & \text{si } h(\ulcorner \varphi_x(x) \urcorner) = 0 \\ \bot & \text{sinon} \end{cases}$$

Comme g est semi-calculable, il existe e tel que φ_e calcul g, soit :

$$\varphi_e(x) = \begin{cases} 0 & \text{si } h(\ulcorner \varphi_x(x) \urcorner) = 0 \\ \bot & \text{sinon} \end{cases}$$

Et en exécutant cette machine de Turing sur l'entrée e, nous avons :

$$\varphi_e(e) = \begin{cases} 0 & \text{si } h(\ulcorner \varphi_e(e) \urcorner) = 0 \\ \bot & \text{sinon} \end{cases}$$

ce qui mène une contradiction car $\varphi_e(e)$ serait définie si et seulement si $\varphi_e(e)$ ne l'est pas.

□

Remarquons que nous avons, pour cette démonstration, utilisé ce que nous pourrions appeler une auto-référence, dans la définition de la fonction φ_x. Il est raisonnable de se demander si une telle procédure est autorisée, et si oui quel est son sens. C'est quelque part la question à laquelle répond le théorème de récursion de Kleene. Ce dernier nous garantissant en effet la possibilité d'usage d'une telle construction.

3 Théorème de Kleene

Lemme 1. *Pour toute fonction semi-calculable $g(x, y)$, il existe un indice e tel que pour tout y,*

$$\varphi_e(y) \simeq g(e, y).$$

Lemme 2. *Pour toute fonction calculable $f(x)$, il existe un indice e tel que pour tout y,*

$$\varphi_e(y) \simeq \varphi_{f(e)}(y).$$

Lemme 3. *Les lemmes 1 et 2 sont équivalentes.*

Démonstration. $\boxed{1 \Rightarrow 2}$ Pour une fonction f donnée, définissons la fonction g par $g(x, y) \simeq Un(f(x), y)$, où Un est la fonction universelle de Kleene. Le lemme 1 nous donne un indice e tel que pour tout y,

$$\begin{aligned} \varphi_e(y) &= g(x, y) \\ &= Un(f(e), y) \\ &= \varphi_{f(e)}(y). \end{aligned}$$

$\boxed{2 \Rightarrow 1}$ Pour une fonction g donnée, le théorème d'itération de Kleene (autrement connu sous le nom de théorème $\mathrm{s_{mn}}$) nous donne f tel que pour tout x et y, $\varphi_{f(x)}(y) \simeq g(x, y)$. Le lemme 1 nous donne un indice e tel que,

$$\begin{aligned} \varphi_e(y) &= \varphi_{f(e)}(y) \\ &= g(x, y). \end{aligned}$$

Ainsi, nous bien avons démontré la double implication et par conséquent l'équivalence $2 \Leftrightarrow 1$. □

En d'autres termes, le lemme 1 nous apprend que pour toute fonction unaire f, il existe un programme p calculant $f(p)$. Ce théorème a été prouvé en 1938 mais seulement publié en 1952 [Kle52].

Théorème 2 (Théorème de récursion de Kleene)**.** *Les lemmes 1 et 2 sont vrais.*

De façon informelle, une manière de démontrer ce théorème serait de constructuire un quine, qui légèrement transformé donnerait la fonction que nous souhaitons construire.

Démonstration. Posons donc dans un premier temps la fonction `diag` qui pour une chaine de caractères `s` donnée en entrée, renvoie celle-ci dans laquelle chaque occurence du caractère `x` est remplacé par la chaine `s` entre guillemets. Par exemple, appliqué à la chaine `"boucle x ici"`, cette fonction rendra le résultat `"boucle "boucle x ici" ici"`. Voici le pseudo-code [1] d'un programme correspondant à cette spécification :

```
programme diag(s : chaine_de_caracteres) {
    res := "";
    quoted_s := '"' ++ s ++ '"';
    pour tout entier i de 1 à s.taille :
        si s[i] = 'x' :
            res := res ++ quoted_s;
        sinon :
            res := res ++ s[i];
    rendre res;
}
```

Ainsi, à partir de ce dernier nous pouvons écrire un programme dont l'effet est de s'afficher lui-même : [2]

```
programme auto_impr() {
  print(diag("programme auto_impr() {\n  print(diag(x));\n}"));
}
```

Nous pouvons aussi définir un programme qui termine si et seulement si son argument est son propre code :

```
programme arret_si_identique (y : string) {
    Si y = diag "programme arret_si_identique (y : string) {
                    Si y = diag x :
                        rendre 1;
                    Sinon : tant que 1 faire {};
                }" :
       rendre 1;
    Sinon : tant que 1 faire {};
}
```

Ce dernier est un exemple, généralisable en une fonction g qui peut prendre en entrée son propre code et y pour en faire quoi qu'on puisse imaginer :

```
programme g (code : string, y : entrée) {
```

1. On utilisera la notation `++` pour l'opérateur infix de concaténation de chaines, `'"'` pour désigner le caractère *simple guillemet* et `s[i]` pour l'accès au `i`-ème caractère de la chaine `s`.

2. Le caractère `\n` est utilisé pour désigner un retour chariot.

```
    Si code = diag "programme g (code : string, y : entrée) {
                      Si code = diag x :
                         // faire quelquechose
                      Sinon : // faire autre chose
                   }" :
        // faire quelquechose
    Sinon : // faire autre chose
}
```

Ce dernier programme conclu la preuve. En effet, en assimilant la fonction `diag` à « utiliser un indice », nous avons réussi à construire la fonction g du lemme 1.

□

Combinateurs. Une idée similaire peut être utilisée pour obtenir un *combinateur de point fixe*. Il s'agit de construire un autre λ-terme k à partir d'un λ-terme g, tel que k soit β-équivalent à gk. Ainsi, en definissant les termes $diag \stackrel{\text{def}}{=} \lambda x.xx$ [3], $l \stackrel{\text{def}}{=} \lambda x.g(diag\ x)$, et $k \stackrel{\text{def}}{=} ll$, nous avons :

$$\begin{array}{rcl} k & = & \big(\lambda x.\ g(diag\ x)\big)\big(\lambda x.\ g(diag\ x)\big) \\ & = & \big(\lambda x.\ g((\lambda y.\ yy)x)\big)\big(\lambda x.\ g((\lambda y.\ yy)x)\big) \\ & \to_\beta & \big(\lambda x.\ g(xx)\big)\big(\lambda x.\ g(xx)\big) \\ & \to_\beta & g\big(\lambda x.\ g(xx)\big)\big(\lambda x.\ g(xx)\big) \\ & \leftarrow_\beta^2 & g\big(\lambda x.\ g((\lambda y.\ yy)x)\big)\big(\lambda x.\ g((\lambda y.\ yy)x)\big) \\ & = & g\big(\lambda x.\ g(diag\ x)\big)\big(\lambda x.\ g(diag\ x)\big) \\ & = & gk \end{array}$$

Si maintenant on prend

$$Y \stackrel{\text{def}}{=} \lambda g.\big((\lambda x.g(xx))(\lambda x.g(xx))\big)$$

on remarque que $Y\ g \equiv_\beta g(Y\ g)$. Cette opérateur est plus connu sous le nom de *combinateur de point fixe de H.Curry* (ou « Y-combinator »). Une autre personnalité de l'informatique a proposé une définition alternative [Tur37], avec

$$\Theta \stackrel{\text{def}}{=} \big(\lambda xy.y(xxy)\big)\big(\lambda xy.y(xxy)\big)$$

nous avons une β-réduction : $\Theta\ g \to_\beta g(\Theta\ g)$ ce qui est plus fort que la β-équivalence. Θ est appelé « combinateur de Turing ».

3. Nous reconnaissons ici l'auto-application, action assimilable à celle du programme `diag` utilisé pour la précédente preuve.

4 Applications

4.1 Quines

Comme expliqué précédemment, un quine est un programme qui lancé sur entrée vide produit entièrement et uniquement une affichage de son propre code source. Par une application directe de la preuve du théorème de Kleene 2 nous pouvons construire un quine dans un langage de programmation. Nous pouvons par exemple en écrire un en langage OCaml, voir Programme 1 ci-après :

Programme 1 Quine en OCaml

```
let diag e =
  let dg = String.make 1 (Char.chr 34) in
  String.split_on_char (Char.chr 120) e
  |> String.concat (dg ^ e ^ dg)

let _ =
  "let diag e =
  let dg = String.make 1 (Char.chr 34) in
  String.split_on_char (Char.chr 120) e
  |> String.concat (dg ^ e ^ dg)

let _ =
  x
  |> diag |> print_endline"
  |> diag |> print_endline
```

Ce quine utilise une fonction très semblable au programme `diag` de la preuve informelle du théorème, à ceci-près qu'il faut y ajouter une gestion des caractère spéciaux pour le type `string`. On y utilise donc des codes ASCII pour les caractères particuliers : *code 34* pour le guillement double (« " ») et *code 120* pour la lettre « x » qui est utilisée strictement de la même façon que le programme `diag` de la preuve.

Remarque Si vous voulez essayer ce quine, le code est téléchargeable à ce lien : `https://www.paulpatault.fr/ocaml-quine.ml`. Les informations pour l'installation d'OCaml sont ici : `https://ocaml.org/docs/install.html`. Enfin, il est possible de tester ce quine avec la simple commande :

```
ocaml ocaml-quine.ml >> output && diff ocaml-quine.ml output
```

Nous constatons alors que le résultat est vide, ce qui indique donc qu'il n'y a pas de

différence entre le code source et le résultat produit.

4.2 Calculabilité

Je n'arrive pas encore a bien expliquer cette section. Travail à faire :
— équivalence des définitions de la notion d'ensemble calculablement énumérable ;
— expliquer théorème 3.5.1 ;
— sans 3.5.1, montrer que la fonction qui étant donné e, renvoie 1 si W_e est calculable et 0 autrement, est incalculable ;
— expliquer la différence entre les deux derniers points.

4.3 Récursivité en programmation

Finalement, ce que nous pouvons tirer de ce théorème est qu'il est possible de définir des fonctions à partir d'elles-même dans des programmes. Cela s'appelle la récursivité (attention, à ne pas confondre avec les *fonctions récursives* de Gödel), c'est la construction `let rec` de OCaml par exemple.

Corollaire 1 (du théorème 2)**.** *Étant données trois fonctions calculables k, l, m ; il existe une fonction semi-calculable f telle que pour tout y,*

$$f(y) \simeq \begin{cases} k(y) & \textit{si } l(y) = 0 \\ f(m(y)) & \textit{sinon.} \end{cases}$$

Alors, nous pouvons définir par exemple la fonction $\gcd(u, v)$ (pour *greatest common divisor*, ce qui correspond à « plus grand diviseur commun ») par cas telle que :

$$\gcd(y) \simeq \begin{cases} k(y) & \text{si } l(y) = 0 \\ \gcd(m(y)) & \text{sinon} \end{cases}$$

$$\begin{aligned} \textit{avec} \qquad y &\stackrel{\text{def}}{=} \langle u, v \rangle \\ k &\stackrel{\text{def}}{=} \Pi_2 \\ l &\stackrel{\text{def}}{=} \Pi_1 \\ m &\stackrel{\text{def}}{=} \big\langle k(y) \ \% \ l(y), l(y) \big\rangle \end{aligned}$$

Le % est l'opération modulo et Π_i est la i-ème projection d'un n-uplet. Les fonctions k,l et m étant calculables et la définition convenant, l'opération $\gcd(u, v)$ est bien calculable.

Références

[Kle52] Stephen Cole Kleene. *Introduction to metamathematics*. North-Holland Publishing, 1952.

[Tho84] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8) :761–763, 1984.

[Tur37] A. M. Turing. The þ-function in λ-κ-conversion. *Journal of Symbolic Logic*, 2(4) :164–164, 1937.