Chapitre 7 : Calculabilité

à lire pendant les vacances de Toussaint

Comme vous le savez déjà, la notion d'algorithme en tant que méthode automatique de résolution existe depuis l'antiquité.

À la fin du XIXème siècle et au début du XXème siècle, une question émerge : est-il possible "d'automatiser" la preuve mathématique ? Cette question est la suite naturelle des efforts de formalisation des mathématiques de la même époque.

En 1900, Hilbert pose le problème suivant (connu sous le nom de *Entscheidungsproblem*) : trouver un algorithme qui décide si un énoncé mathématique est vrai.

La notion d'algorithme étant floue, un premier travail consistera à formaliser cette notion (Turing - Church - Herbrand et Gödel) puis à donner une réponse négative à l'*Entscheidungsproblem* en mettant en évidence que l'algorithme cherché n'existe pas. Suite à ce résultat, la théorie de la calculabilité qui consiste à distinguer les problèmes pour lesquels on peut construire un algorithme de ceux pour lesquels ce n'est pas possible, se développe.

Pour pouvoir étudier formellement ces questions il est nécessaire de poser des définitions précises : qu'entend-on par *problème*, *algorithme* ? Le programme ne souhaite pas entrer dans le détail de ces questions qui demeurent très formelles en faveur d'une première approche plus intuitive mais il faut avoir en tête qu'un formalisme rigoureux existe.

1 Problème de décision

Nous allons commencer par préciser quel type de *problème* nous allons considérer : nous traiterons uniquement dans ce chapitre la notion de problème de décision. Un problème de décision est spécifié sous la forme suivante : on se donne une entrée et on doit simplement décider si elle satisfait une certaine propriété ou non.

Plus formellement, un problème de décision sur un domaine d'entrées D est la donnée d'une fonction booléenne totale sur D. On dit qu'un algorithme résout un problème de décision associé à la fonction $f : D \to \mathbb{B}$ lorsque pour chaque élément $e \in D$, il renvoie le booléen $f(e)$. Un élément e de D est une instance du problème et tout élément dont l'image est `true` est appelée instance positive du problème alors que les autres sont appelées instances négatives.

Exemple 1. *Le problème suivant est un problème de décision :*
- *Entrée : un graphe fini non orienté G.*
- *Sortie* `true` *si G est connexe et* `false` *sinon.*

Dans la notion de problème que nous avons considéré ici, la nature des arguments passés en entrée est quelconque. Cependant, les objets que la machine peut manipuler sont en bijection avec l'ensemble des chaînes de caractères. Un problème de décision correspondra donc à une fonction totale booléenne dont le domaine de définition est dénombrable.

2 Un modele de calcul

La définition usuelle d'un algorithme est essentiellement "une suite finie d'instructions élémentaires".

Comme mentionné en introduction, des efforts considérables ont été menés au début du XXème siècle pour donner une définition formelle de la notion d'algorithme. Trois modèles distincts mais mathématiquement équivalents ont été développés : les machines de Turing, le λ-calcul de Church et les fonctions récursives d'Herbrand - Gödel. Ces trois modèles sont encore aujourd'hui considérés comme une définition satisfaisante de la notion de programme informatique.

Dans le cadre de notre programme, nous allons considérer qu'un algorithme est un programme écrit en C ou en Ocaml (ou dans votre langage préféré) avec une mémoire infinie (ce qui n'est pas possible dans une architecture matérielle donnée mais doit être satisfait par notre modèle de calcul). Cela signifie que la pile d'appels ne déborde jamais : on peut empiler autant d'appels imbriqués que l'on souhaite et que l'on peut associer autant de mémoire que l'on souhaite pour créer des variables (même des variables allouées ou statiques).

On remarque qu'un programme donné utilise une quantité de mémoire finie mais on veut ici se prémunir d'une limitation de la mémoire qui impliquerait qu'un problème puisse être considéré comme non résoluble

car il nécessite plus de mémoire que ce que notre machine nous fournit. Ceci ne serait pas satisfaisant car on souhaite une définition de la notion de calcul automatique indépendante des problématiques matérielles qui évoluent. La quantité de mémoire est par ailleurs un paramètre dont l'évolution est très importante. On ne veut pas un modèle qui dépend de l'architecture d'une époque donnée mais qui capture une idée plus "absolue". On retiendra en conséquence que bien que décidable, un problème peut ne pas admettre de programme le résolvant sur une machine donnée si celle-ci n'a pas la mémoire suffisante mais, quitte à augmenter cette mémoire, un tel programme existera.

3 Machine universelle-codage

Un point clé de toute notre théorie est qu'un algorithme peut prendre en entrée tout type d'objets et notamment un autre algorithme décrit par exemple par la chaîne de caractères de son code source. C'est quelque chose de crucial car par exemple votre compilateur est de fait un programme qui prend en entrée le programme que vous avez rédigé.

On appelle (et on admet l'existence) machine universelle, un programme qui est capable de simuler tous les autres programmes. Plus précisemment :

Définition 1. *Il existe un programme en OCAML (par exemple)* `u : string -> string-> string` *qui prend en entrée la description d'un programme et la description de son entrée et qui renvoie la description de la sortie du programme sur l'entrée si elle existe et ne se termine pas sinon (si l'exécution du programme sur l'entrée ne termine pas).*

La fonction passée en argument devra être de type `string->string` *ce qui peut être fait sans perte de généralité à codage prêt des ensembles des entrées et des sorties.*

Pour la suite on considérera que les programmes sont décrits par des chaînes de caractères et manipulent et renvoient des chaînes de caractères. En effet une construction propre de la notion de calculabilité passe par une utilisation d'un type d'entrée uniformisé ce qui est possible via un codage (en binaire par exemple ou sur Σ^* où Σ est fini). On remarque alors notamment l'importance de la sérialisation des types sophistiqués (arbre, graphe) vue en première année.

4 Décidabilité

Définition 2. *Un problème de décision associé à $f : D \to \mathbb{B}$ est dit décidable s'il existe un algorithme A spécifié de la manière suivante :*

- *Entrée : $e \in D$.*
- *Sortie : $f(e)$.*

La fonction f associée est dite calculable.

Si un tel algorithme n'existe pas, on dit que le problème est indécidable.

Exemple 2. *Si D est un ensemble fini alors le problème de décision associé à f définie sur D est décidable.*

Exemple 3. *Montrer que le problème de savoir si un automate fini reconnait le langage vide est décidable.*

Définition 3. *Le problème de l'arrêt est défini de la manière suivante :*

- *Entrée : un algorithme A et une entrée e,*
- *Sortie : oui si l'exécution de A sur e termine en temps fini et non sinon.*

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

Supposons qu'il existe une fonction en C : `bool halts(char* prog, char* e)`. On suppose que le programme et l'entrée sont donnés sous forme de chaînes de caractères ce qui peut être supposé sans perte de généralité. On va alors construire un programme `paradox.c` dont le comportement ne peut pas être correctement spécifié ce qui garantira par l'absurde que la fonction `halts` n'existe pas.

Voici le contenu de `paradox.c` :

```
void main(int argc, char** argv){
   FILE* f = fopen("paradox.c", "r");

   char* prog = malloc(sizeof(char)*BCP);
   int i=0;
   while (fscanf(f,"%c",&prog[i])!=EOF){
      i++;
   }
   prog[i]='\0';
   if (halts(prog, argv[1])){
      while(true) {}
   }
}
```

Que va donner la commande `gcc paradox.c -o paradox` suivie de `./paradox bla` ?

Résumé de l'idée :

Théorème 2. *Il existe une infinité de fonctions au sens mathématiques de la forme $f : D \to E$ pour lesquelles il n'existe pas d'algorithme qui produise en sortie $f(x)$ sur chaque entrée $x \in D$. Ces fonctions sont dites non calculables.*