Calculabilité

Montaigne 2023-2024

– mpi23@arrtes.net –

Cadre d'étude

Modèle de calculs

Les ordinateurs **calculent**, c'est-à-dire qu'ils réalisent des séquences d'opérations décrites par un **algorithme**, pour produire un résultat à partir de certaines entrées. De ce point de vue, un algorithme est une **description finie** d'un ensemble, potentiellement infini, d'étapes à exécuter.

La variété des algorithmes permet aux ordinateurs d'accomplir des tâches multiples. Mais des **limites indépassables** et **indépendantes des technologies** à ce qu'un algorithme peut exprimer, et donc à ce qu'un ordinateur peut calculer, existent.

Modèle de calculs

Dans ce chapitre, un **algorithme** est un **programme** écrit dans l'un quelconque des deux langages C ou OCaml.

Notre objectif est de caractériser les **problèmes** qui peuvent, ou non, être résolus par un **algorithme** (= un **programme**).

Pour établir des résultats généraux, on suppose que les programmes s'exécutent sur un **ordinateur idéal** doté d'une **mémoire illimitée**.

En pratique, chaque exécution d'un programme n'utilise jamais qu'une quantité finie de mémoire. Mais on ignore simplement les limites que poserait une machine physique dont la quantité de mémoire utilisable est finie.

Modèle de calculs

Un **modèle de calcul** est défini comme un programme C ou OCaml, s'exécutant sur une machine ayant les deux propriétés idéales suivantes.

- La **pile d'appels** ne déborde jamais, quel que soit le nombre d'appels de fonctions emboîtés.
- Il n'existe **aucune limite à la quantité de données** pouvant être allouées sur le **tas**.

Pourquoi une mémoire illimitée ?

En ne limitant pas la taille de la mémoire, on s'affranchit des contraintes techniques liées à des systèmes construits à une époque donnée et on s'autorise à manipuler, en machine, des données de taille arbitrairement grande.

Ce qui permet de mener des raisonnements asymptotiques[1] sur des entrées arbitrairement grandes.

Un **ordinateur idéal** fait donc abstraction des limites de la mémoire physique de l'**ordinateur réel**.

Bien évidemment, étant donné un programme résolvant un problème donné, il est entendu que, sur chaque **ordinateur réel**, ce programme ne résout effectivement le problème que sur les entrées de taille adaptée à la machine.

1. Étude de complexité.

Entrées et sorties

Les **entrées** et les **sorties** des programmes peuvent **a priori** être de n'importe quel type du langage utilisé.

Dans ce cours, les **entrées** sont représentées par des **chaînes de caractères**, type apte à représenter n'importe quelle donnée de n'importe type manipulable par un ordinateur.

Cette restriction ne réduit en rien la portée de l'étude et le choix de ce type unique écarte quelques difficultés.

Problème de l'arrêt

Formulation du problème

Étant donnés un algorithme A et une entrée e pour cet algorithme, le **problème de l'arrêt** consiste à déterminer si l'exécution de A sur e s'arrête après un nombre **fini** d'étapes.

Résoudre ce problème, c'est fournir un programme qui prend en entrée une paire (A, e) et qui renvoie **en un temps fini** un booléen indiquant si l'exécution de A sur e termine ou non.

L'énoncé de ce problème est formalisé ci-après en OCaml mais le C aurait convenu tout aussi bien.

Formulation du problème

Définition 1 (problème de l'arrêt en OCaml)

Le **problème de l'arrêt** consiste à écrire une fonction OCaml :

```
halt: string -> string -> bool
```

prenant en entrées :

- une chaîne `f` contenant le code source d'un programme OCaml,
- une chaîne `e` représentant des entrées pour le programme donné par `f`,

et qui, pour toutes chaînes `f` et `e`, termine en un temps fini en renvoyant `true` si l'exécution du programme `f` sur les entrées `e` termine en temps fini, et `false` sinon.

Indécidabilité du problème

Ce problème n'a pas de solution algorithmique : il est **indécidable**.

Théorème 2

Il n'existe pas de fonction OCaml résolvant le problème de l'arrêt.

Démonstration

Ce résultat se démontre par l'absurde, en supposant l'existence d'une fonction `halt: string -> string -> bool` répondant à la spécification du problème de l'arrêt et en s'en servant pour construire un programme dont le comportement est contradictoire.
Supposons l'existence d'une telle fonction `halt` et considérons alors la fonction `weird` suivante.

```
let weird f e =
  if halt f e then
    while true do print_string "ok" done;
```

Indécidabilité du problème

Démonstration

Cette fonction reçoit les arguments `f` et `e` puis teste, par l'appel à la fonction `halt`, l'arrêt de l'exécution du programme `f` sur les entrées `e`.

- Si cet appel renvoie `true`, c'est que l'exécution de `f` sur `e` s'arrête effectivement. La fonction `weird` entre dans une boucle infinie et ne s'arrête jamais.
- Si cet appel renvoie `false`, c'est que l'exécution de `f` sur `e` ne s'arrête pas. La fonction `weird` ne fait alors rien et s'arrête immédiatement.

Ainsi `weird f e` s'arrête si et seulement `halt f e` ne s'arrête jamais.
Considérons à présent la fonction `paradox`.

```
let paradox g = weird g g
```

Alors `paradox f` s'arrête uniquement si et seulement si l'exécution de `f` sur `f` ne s'arrête jamais.
Appliquons cette fonction à elle-même. En vertu de ce qui vient d'être établi, `paradox paradox` s'arrête si et seulement si `paradox paradox` ne s'arrête jamais !
Résultat qui établit clairement une équivalence entre une proposition et sa négation et constitue, de fait, une contradiction.
En conséquence, l'existence de la fonction `paradox` est contradictoire et, en remontant la chaîne des fonctions ayant mené à sa définition, celle de la fonction `halt` l'est aussi.
Il n'existe donc pas de fonction `halt`. Le problème de l'arrêt est **indécidable**.

Bien que simple à énoncer, le problème de l'arrêt illustre une situation particulière qui n'admet aucune solution algorithmique.

La suite de ce chapitre formalise ces idées pour caractériser tous les problèmes qui peuvent ou ne peuvent pas être résolus par des algorithmes.

Problèmes de décision

Notion de problème

Caractériser un **problème algorithmique**, c'est se donner une **spécification** sous la forme :

- un ensemble d'**entrées** admissibles,
- une description des **résultats attendus**, en fonction de l'entrée.

Ainsi, un problème algorithmique s'appliquant à des entrées prises dans un ensemble E et produisant des résultats (ou **sorties** ou **solutions**) pris dans un ensemble S peut être traduit par une relation binaire $\mathcal{R} \subseteq E \times S$ associant chaque entrée e aux résultats admissibles pour cette entrée e.

Notion de problème

Un cas particulier important est celui où cette relation $\mathcal{R}$ définit en réalité une fonction, c'est-à-dire où il ne peut y avoir qu'un seul résultat admissible par entrée.

Un aspect clé d'un problème algorithmique est qu'il s'applique à un ensemble **a priori** infini d'entrées possibles.

Résoudre un problème algorithmique, c'est proposer un algorithme qui produit un résultat correct pour chacune des entrées possibles.

Fonction calculable

Le mot **fonction**, selon le contexte, couvre au moins deux notions différentes.

- Une **fonction mathématique** est une relation binaire entre des antécédents et des images, qui associe au plus une image à chaque antécédent.
- Dans un programme, une **fonction** est un fragment de code décrivant les opérations qui permettent de produire un résultat à partir d'une entrée. Pour éviter les ambiguïtés, une telle fonction est appelée **algorithme** dans la suite du cours.

Fonction calculable

La notion mathématique de fonction explicite ce qu'est l'image de chaque élément d'un ensemble potentiellement infini d'entrées, sans nécessairement décrire la manière dont chacune peut être calculée. On parle de définition **extensionnelle**.

À l'inverse, un algorithme est précisément une méthode de calcul, exprimée par un texte fini : son code. On a cette fois une définition **intensionnelle**. À tout algorithme, on peut associer une fonction mathématique, liant les entrées du programme aux sorties qu'il calcule.

On dit que l'algorithme **réalise** une fonction mathématique, et la fonction mathématique réalisée par un algorithme est appelée la **sémantique** de l'algorithme. En revanche, une fonction mathématique peut ne pas admettre de réalisation par un programme.

Fonction calculable

On dit qu'une fonction mathématique est **calculable** s'il existe un algorithme qui la réalise.

Définition 3 (fonction calculable)

Une fonction mathématique totale $f : E \to S$ est **calculable** s'il existe un algorithme A tel que, pour toute entrée $e \in E$, l'algorithme A appliqué à e produit le résultat $f(e)$, en un temps fini.

Montrer qu'une fonction mathématique est calculable est en un sens assez simple : il **suffit** de fournir un procédé de calcul effectif, par exemple sous la forme d'un programme C ou OCaml, que l'on pourra appeler **algorithme**.

C'est d'ailleurs exactement ce qui s'est passé avec l'essentiel des questions abordées jusqu'ici.

Fonction non calculable

Montrer qu'un problème algorithmique **ne peut pas** être résolu est nettement plus délicat : il faut alors montrer qu'**il n'existe aucune** manière effective de réaliser le calcul demandé. Partant de notre habitude de résoudre les questions algorithmiques à l'aide de programmes, nous devons résoudre au moins deux questions délicates.

- D'abord une question purement technique : par quelle méthode démontrer qu'il n'existe aucun programme réalisant une fonction mathématique donnée ?
- Ensuite, à supposer que l'on ait pu conclure positivement à la question précédente, peut-on réellement en déduire qu'il n'existe aucun procédé de calcul ? Ou avons-nous seulement mis en lumière une limitation du langage de programmation choisi ?

Fonction non calculable

Le problème de l'arrêt illustre la première question. Le traitement de la seconde question est différé.

Avant cela, montrons l'existence de nombreuses fonctions non calculables par un simple argument de cardinalité.

Théorème 4 (existence de fonctions non calculables)

Il existe une infinité de fonctions non calculables.

Démonstration

Considérons les fonctions mathématiques de $\mathbb{N}$ dans $\mathbb{B}$. Leur ensemble est infini, et même non dénombrable (théorème de Cantor).
L'ensemble des programmes, bien qu'infini également, est à l'inverse dénombrable, car chaque programme est défini par une chaîne de caractères (son code). Il existe donc moins de programmes que de fonctions mathématiques et toutes les fonctions ne peuvent pas être réalisées.

Problèmes de décision

Souvent, on s'intéresse à un type de problème particulier : les questions à propos de l'entrée dont la réponse est **oui** ou **non**. Ces problèmes sont appelés des **problèmes de décision** et comprennent en particulier le problème de l'arrêt.

Définition 5

Un **problème de décision** sur un domaine d'entrées E est défini par une fonction totale $f : E \to \mathbb{B}$.

Un algorithme A **résout** un problème de décision f si, pour toute entrée $e \in E$, l'algorithme A appliqué à e termine en un temps fini et produit le résultat $f(e)$.

Problèmes de décision

Chaque élément $e \in E$ du domaine d'entrées est appelé une **instance** du problème.

De manière équivalente, un problème de décision sur un domaine d'entrées E peut être défini par un ensemble P d'**instances positives** : l'ensemble des $e \in E$ tels que $f(e) = \mathrm{V}$.

Problèmes de décision

Par nature, un problème algorithmique ne s'applique pas à une entrée particulière, mais à tout un domaine. Ainsi, la question suivante, bien qu'intéressante et pas si simple, **n'est pas** un problème algorithmique. En effet, elle s'applique à une configuration de départ de ce jeu très précise, et on y répond par le simple mot **oui**.

Y a-t-il une solution au problème de l'âne rouge ?

Le problème algorithmique associé consisterait à généraliser la question, par exemple sous la forme :

Y a-t-il une solution au problème de l'âne rouge à partir d'une configuration arbitraire donnée en entrée ?

Résoudre ce problème consisterait alors à fournir un algorithme permettant, pour chaque plateau de jeu possible, de déterminer s'il existe une solution en partant de cette configuration.

Problèmes de décision

Dans ce chapitre, on suppose systématiquement que le domaine E des entrées possibles est **infini** et **dénombrable**. La contrainte de rester dans le domaine du dénombrable se déduit de la nécessité d'avoir une représentation finie des données.

En effet, il est impossible de représenter les éléments d'un ensemble indénombrable par des chaînes de caractères (ou de bits) finies. Et pour cause : l'ensemble de ces chaînes est dénombrable !

Problèmes de décision

À l'inverse, les problèmes sur un domaine fini sont peu intéressants pour la **théorie de la calculabilité** : si on s'intéresse à une question portant sur un domaine fini, on peut, du moins en théorie, énumérer toutes les entrées possibles, déterminer à l'avance le résultat attendu pour chacune, puis proposer l'**algorithme** qui se contente de consulter la table des résultats précalculés pour répondre immédiatement. Sur un domaine infini, en revanche, cette énumération exhaustive est impossible et il devient nécessaire de déterminer un procédé permettant d'obtenir la réponse y compris sur des entrées que nous n'avons pas étudiées préalablement.

La question :

Y a-t-il une solution au problème de l'âne rouge à partir d'une configuration arbitraire donnée en entrée ?

n'a pas un domaine infini si on se limite aux configurations de départ pouvant être obtenues avec les dimensions du plateau classiques. Le domaine devient en revanche infini dès lors que l'on admet des plateaux de jeu de taille arbitraire. On parle en général ici de **jeu généralisé**.

Décidabilité

Décidable / Indidécidable

Un problème de décision est **décidable** lorsqu'il peut être résolu par un algorithme, c'est-à-dire lorsque la fonction sous-jacente est calculable.

Définition 6 (problème décidable)

Un problème de décision défini par une fonction $f : E \to \mathbb{B}$ est **décidable** si f est calculable.

Il existe donc un algorithme A tel que, pour toute entrée $e \in E$, l'algorithme A appliqué à e termine en un temps fini et renvoie V si $f(e) = \text{V}$ et F sinon. On dit qu'un tel problème est **décidé** par l'algorithme A.

Un problème **indécidable** est un problème de décision qui n'est pas décidable, c'est-à-dire qui ne peut pas être résolu par un algorithme.

Exemples

Les problèmes de décision suivants sont décidables.

- Y a-t-il un doublon dans un tableau a d'entiers ?
  Entrées : tableaux d'entiers, sans restriction de taille.
- Un nombre n est-il premier ?
  Entrées : nombres entiers positifs, sans restriction.
- Y a-t-il un chemin entre les sommets s et t d'un graphe g ?
  Entrées : triplets formés d'un graphe et de deux sommets.
- Un graphe g peut-il être colorié avec quatre couleurs ?
  Entrées : graphes non orientés.
- Un mot m est-il accepté par un automate fini a ?
  Entrées : paires formées d'un automate fini et d'un mot.
- Une grammaire algébrique $\mathcal{G}$ peut-elle générer le mot vide ?
  Entrées : grammaires algébriques.

Semi-décidabilité

Une caractéristique importante d'un algorithme **décidant** un problème algorithmique est qu'il termine sur toute entrée en renvoyant Vou F. Autrement dit, la sémantique d'un tel algorithme est une fonction totale. On obtient une version amoindrie de décidabilité en assouplissant ce critère.

Définition 7 (problème semi-décidable)

Un problème de décision défini par une fonction $f : E \to \mathbb{B}$ est **semi-décidable** s'il existe un algorithme A tel que, pour toute entrée $e \in E$, l'algorithme A appliqué à e :

- termine en un temps fini et renvoie Vsi $f(e) = \mathrm{V}$,
- renvoie F, **ou ne termine pas**, **ou échoue**, sinon.

Semi-décidabilité

Un algorithme de semi-décision doit nécessairement pouvoir répondre positivement pour toute instance positive, mais peut ne pas terminer sur les instances négatives. Par conséquent, lorsqu'un algorithme de semi-décision prend du temps à répondre, cela peut signifier deux choses :

- soit il a simplement besoin de plus de temps pour conclure (et l'instance peut être positive)
- soit il est en train de diverger (et l'instance serait alors négative).

Semi-décidabilité

La dernière voie d'exécution possible pour un programme à valeur de retour booléenne est d'échouer, ce qui se manifeste par exemple par une interruption liée à une exception non rattrapée. À l'instar de la divergence, cette issue n'est acceptable dans une procédure de semi-décision que pour les instances négatives.

De nombreux problèmes indécidables « naturels » sont en réalité semi-décidables. C'est le cas par exemple du problème de l'arrêt.

En revanche, le même argument de cardinalité qui établissait l'existence d'une infinité de problèmes indécidables permet encore de déduire qu'il existe une infinité de problèmes qui ne sont pas semi-décidables. Les contre-exemples sont moins naturels, et tendent à être construits à l'aide de **constructions diagonales**.

Un exemple de problème non semi-décidable

Théorème 8

Considérons le problème de décision consistant à déterminer à partir du code source d'une fonction de type `string -> bool`*, si cette fonction appliquée à son propre code source ne renvoie pas* `true`*. Ce problème n'est pas semi-décidable.*

Démonstration

Supposons que ce problème soit semi-décidable. Il existe donc une fonction OCaml `diag: string -> bool` avec le comportement suivant :

- si `s` est le code source d'une fonction OCaml `f: string -> bool` telle que `f s` renvoie false, alors `diag s` renvoie true,
- si `s` est le code source d'une fonction OCaml `f: string -> bool` telle que `f s` ne termine pas ou échoue, alors `diag s` renvoie true,
- si `s` est le code source d'une fonction OCaml `f: string -> bool` telle que `f s` renvoie true, alors `diag s` soit renvoie false, soit ne termine pas, soit échoue.

Un exemple de problème non semi-décidable

Démonstration

Considérons l'application de `diag` à son propre code source `s`. On énumère alors les trois cas précédents :

- si `diag s` renvoie false, alors `diag s` renvoie true,
- si `diag s` ne termine pas ou échoue, alors `diag s` renvoie true (en un temps fini),
- si `diag s` renvoie true (en un temps fini), alors `diag s` renvoie false, ou ne termine pas, ou échoue.

Dans chacun de ces cas, le comportement de `diag s` est contradictoire. La fonction `diag` ne peut donc pas exister.

Algorithme universel

Existence d'un algorithme universel

Dans les premiers exemples de problèmes indécidables rencontrés jusqu'ici, un algorithme peut prendre en entrée un autre algorithme, par exemple donné par son code source.

Ceci n'a rien d'extraordinaire. En programmation, certains **programmes peuvent prendre en entrée d'autres programmes**. Par exemple, les compilateurs C et OCaml sont des programmes prenant en entrée un fichier source et produisent un fichier exécutable équivalent.

Existence d'un algorithme universel

Dans ce cadre, on appelle **algorithme universel**, ou **machine universelle**, un algorithme capable de simuler tous les autres, c'est-à-dire un algorithme U qui s'applique à un algorithme A et une entrée e pour A, et qui simule l'action de A sur e.

Théorème 9 (admis)

Il existe une fonction OCaml :

```
eval: string -> string -> string
```

qui prend en entrée le code source `s` *d'une fonction OCaml :*

```
f: string -> string
```

et un argument `e` *de type* `string` *pour* `f`*, et telle que :*

- `eval s e` *termine et renvoie la valeur produite par* `f e` *si l'exécution de* `f e` *termine,*
- `eval s e` *ne termine pas si* `f e` *ne termine pas.*

Existence d'un algorithme universel

L'énoncé de ce théorème utilise `string` comme type de sortie puisque celui-ci permet de représenter n'importe quelle donnée. En langage moderne, un tel algorithme universel est appelé un **interprète**. Un tel interprète se trouve justement au cœur de la boucle interactive[2] d'OCaml.

On retiendra deux étapes principales dans son fonctionnement :

1. l'analyse syntaxique de la chaîne donnée en paramètre[3] pour construire son arbre de syntaxe abstraite,
2. l'évaluation de l'expression ainsi décodée.

2. Toplevel.
3. Voir le cours sur les langages.

Réduction calculatoire