L'interprétation des langages de programmation est quelquefois définie de façon imprécise. Un même programme peut, selon les compilateurs – les logiciels qui traduisent les programmes en données assimilables par la machine –, donner des résultats différents. Pour lever ces ambiguïtés, Dana Scott et Christopher Strachey conçurent, en 1969, des méthodes fondées sur les points fixes. Depuis, l'usage des points fixes s'est étendu aux domaines de l'intelligence artificielle et du génie logiciel (bases de données, programmation, vérification des programmes…).

Commençons ce tour d'horizon des applications du point fixe en informatique par l'interprétation des langages de programmation. On a souvent recours dans la conception de programme à des boucles d'itération. Par exemple, le calcul d'un nombre n à la puissance q consiste à itérer q fois la multiplication par n : $n^q = (((1 \times n) \times n) \ldots \times n)$. Chacune des parenthèses représente une étape d'une boucle d'itération. Pour compter le nombre d'étapes effectuées, on associe à la boucle une variable incrémentée à chaque réalisation d'une étape. Il peut se faire que cette variable interfère avec d'autres variables internes à la boucle. Illustrons cette idée avec l'exemple d'une boucle chargée d'afficher à l'écran deux valeurs, l'une à la suite de l'autre : 3, puis 6. Pour montrer les problèmes d'interprétation des compilateurs, nous avons donné le même nom, *i*, à la variable compteur et à la variable interne à la boucle :

{*i* = 0
for *i* = 1 to 2 do (*i* est ici une variable compteur)
i : = 3*i* ; (*i* comme variable interne à la boucle)
print *i* ;
end do}

Pour exécuter cette boucle, certains compilateurs interpréteront le *i* compteur et le *i* interne à la boucle comme deux variables distinctes. Les compilateurs qui fonctionneront ainsi afficheront les bonnes valeurs : 3, puis 6. D'autres compilateurs, en revanche, assimileront les deux variables à une seule donnée, ne réaliseront la boucle qu'une seule fois, et n'afficheront que le premier des deux chiffres, 3. Pour cette raison, ce type de boucle est à éviter. Néanmoins, des informaticiens ont cherché à élaborer des langages, tel *Ada*, dont les compilateurs ne seraient pas confrontés à ces dilemmes. La solution retenue fut de définir de façon univoque les boucles à l'aide de points fixes.

Reprenons l'exemple du calcul de la puissance n^q. À ce calcul, faisons correspondre un point fixe d'une certaine transformation T. Ce point fixe n'est pas un nombre, mais une fonction (dans la suite, nous verrons qu'en informatique, les points fixes peuvent aussi être des ensembles). La transformation T associe à une fonction $f(n, q)$ des variables n et q, la fonction qui vaut 1 si q est nul, et $n \times f(n, q–1)$ sinon. La fonction n^q est bien un point fixe de la transformation T : $T(n^q)$ est égal à 1 si $q = 0$, à $n \times n^{q-1}$, soit n^q, sinon, ce qui correspond bien aux valeurs de n^q quelle que soit la valeur que prend q.

Le résultat n^q est donc bien un point fixe ; pour remplacer la boucle, nous allons examiner comment l'atteindre. Appliquons la transformation T un certain nombre de fois (q + 1 fois pour être exact) à la fonction vide :

$$T\,[If\,](n,q) = \begin{cases} 1 \text{ si } q = 0 \\ n \times f(n, q-1) \end{cases}$$

$$T\,[\varnothing](n,q) = \begin{cases} 1 \text{ si } q = 0 \\ \varnothing \text{ sinon} \end{cases}$$

$$T^2\,[\varnothing](n,q) = \begin{cases} 1 \text{ si } q = 0 \\ n \times T\,[\varnothing](n, q-1) \text{ sinon} \end{cases} = \begin{cases} 1 \text{ si } q = 0 \\ n \times \begin{cases} 1 \text{ si } q = 1 \\ \varnothing \text{ sinon} \end{cases} \end{cases}$$

$$= \begin{cases} 1 \text{ si } q = 0 \\ n \text{ si } q = 1 \\ \varnothing \text{ sinon} \end{cases}$$

$$\vdots$$

$$T^{n+1}\,[\varnothing](n,q) = \begin{cases} 1 \text{ si } q = 0 \\ n^q \text{ sinon} \end{cases} = n^q$$

Nous voyons, à travers l'exemple de cette fonction n^q, que les points fixes définissent des fonctions par une boucle qui ne présente pas d'ambiguïté sur l'utilisation des indices. Les points fixes donnent un «sens mathématique» aux boucles. Précisons que le programmeur ne change pas ses habitudes de programmation ; il continue à écrire ses boucles comme précédemment ; l'utilisation de point fixe ne concerne que le programmeur en charge du compilateur qui transforme les boucles en points fixes.

Les points fixes de ***bases***

Les points fixes facilitent la construction et l'utilisation de bases de données de taille variable. Par exemple, afin de répondre aux demandes de leurs clients désirant se rendre d'une ville à une autre, les compagnies aériennes ont déterminé, pour chacune des villes qu'elles desservent, les liaisons possibles avec les autres villes. Pour stocker cette information dans des fichiers, il faudrait construire ces fichiers en prenant une par une les villes desservies et en vérifiant s'il existe un vol les connectant avec chacune des autres villes, par autant d'intermédiaires que nécessaires. Cette méthode présente l'inconvénient de devoir recommencer l'intégralité de la procédure chaque fois que la compagnie aérienne ajoute une ville à sa liste de destinations.

Cet inconvénient disparaît avec les méthodes de points fixes. Prenons le cas d'une compagnie aérienne qui dessert quatre aéroports, Paris, Nantes, Nice et Bordeaux, et propose les liaisons aériennes (Nantes, Paris), (Paris, Nice), et (Paris, Bordeaux) :

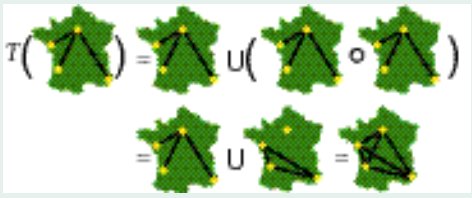
D'après cette liste, le trajet Nantes - Nice est, parmi d'autres, possible via Paris. Notre objectif est de disposer de l'ensemble des trajets, y compris ceux avec correspondance. Ici, les objets mathématiques auxquels sont appliquées les transformations sont des ensembles. Comme nous le verrons, la transformation T transforme une liste en la réunion de la liste des vols directs et de la liste obtenue par composition des vols directs et de cette liste.

Par composition, on entend la mise en relation des villes jointes par une ville intermédiaire : la composition

– notée ∘ – du couple de villes (X, Y) avec le couple (Y, Z) est le couple (X, Z) :

Le point fixe de *T* est la liste où figurent tous les vols possibles. Il est obtenu en un nombre d'itérations inférieur au nombre de villes. Selon la définition de T, nous écrivons :

Nous appliquons la transformation *T* au premier résultat pour aboutir au point fixe :



En calculant l'image de cette liste par la transformation, on trouverait qu'elle est égale à elle-même, ce qui nous prouve qu'elle représente bien le point fixe de *T*. Ici deux itérations suffisent : le point fixe obtenu comporte bien toutes les liaisons aériennes possibles.

Les opérations que nous venons d'effectuer sur une liste à trois couples de villes ne dépendent pas du nombre total de villes. Pour un nombre supérieur à trois couples, le principe resterait le même, le seul changement étant le nombre d'itérations de la transformation pour atteindre le point fixe. Avec cette méthode, le programmeur qui met à jour la base de données en incorporant une nouvelle destination se contente d'ajouter la ville correspondante à la liste originelle ainsi que les villes qui lui sont reliées par un vol direct. Nul besoin d'implémenter le logiciel de réservation pour qu'il tienne compte d'une nouvelle ville, ni de calculer ou stocker toutes les villes qui lui sont reliées. Lorsqu'un client demandera à aller d'une ville à une autre, le logiciel calculera les divers trajets possibles en utilisant la base de données enrichie et la transformation *T*.

Les points fixes vérificateurs

Le but de la vérification est de spécifier, puis vérifier les propriétés des programmes. Imaginons un ensemble de supermarchés d'une même chaîne qui chaque soir mettent à jour leur stock local, puis passent leur commande à l'entrepôt central. Pour passer commande, le supermarché doit établir une liaison, téléphonique ou informatique, avec sa centrale. Il peut y avoir plusieurs échecs avant d'obtenir la connexion. Pour vérifier le bon fonctionnement du système, comme on ignore le nombre d'échecs, il faut envisager les cas de figures où la liaison s'établit après un, deux, trois, … essais. Il est irréaliste d'étudier l'infinité des cas. Les méthodes de point fixe permettent de surmonter cette difficulté. Elles reviennent à considérer chaque nouvel essai comme un rapprochement abstrait vers un point fixe. La transformation considérée ici est la tentative de connexion, et l'objet mathématique auquel on applique cette transformation est l'état de l'ordinateur, connecté ou non. En procédant ainsi, on élimine le problème posé par le nombre inconnu et arbitrairement grand d'échecs de connexion: ce nombre est représenté par le point fixe lui-même.

Par ailleurs, tel Prométhée qui fut condamné à voir éternellement son foie mangé par l'aigle le jour, puis régénéré la nuit, le supermarché répète l'opération «connexion, puis commande» tous les soirs. Un autre point fixe représentera cette répétition perpétuelle. D'une certaine manière, les points fixes permettent d'éliminer la référence explicite au temps, car ils synthétisent l'évolution dans le temps de tous les calculs qui convergent vers le point fixe. Ce type de méthode est utile pour la vérification de systèmes complexes synchronisés. Par exemple, sous la direction de André Arnold, de l'Université de Bordeaux, des logiciels de compteurs électriques à télérelève (relevés à distance) ont été vérifiés et améliorés.

Le Banach des informaticiens

Ces exemples nous ont montré comment tirer parti des propriétés du point fixe, mais nous avons omis de dire dans quel cas une transformation admet un point fixe. Le résultat fondamental est le théorème du point fixe de Knaster-Tarski, démontré en 1942 par Tarski (1902 - 1983), et publié en 1955. Le théorème de Tarski est l'équivalent, pour les informaticiens, du théorème de Banach des mathématiciens. Esquissons ce théorème. Les objets naturels en informatique sont ordonnés ; par exemple, les ensembles sont ordonnés par inclusion, un ensemble est plus grand qu'un autre s'il le contient. Le théorème de Tarski assure que, sous certaines conditions, si une transformation *T* préserve l'ordre, *T* a un plus petit point fixe et un plus grand point fixe. Ainsi, l'exemple du supermarché met en jeu deux points fixes : un plus petit point fixe (qui réalise la connexion) imbriqué dans un plus grand point fixe (qui réalise la répétition perpétuelle du protocole «connexion + commande»). En général, un plus petit point fixe représente un événement qui finira par arriver un jour, et un plus grand point fixe représente un événement qui se répète sans fin. Des combinaisons arbitraires de plus grands et plus petits points fixes peuvent être utiles en vérification. Comment identifier la meilleure combinaison et la calculer efficacement? C'est l'un des principaux thèmes de recherche actuels en vérification.

Irène GUESSARIAN,
Professeur à l'Université Paris 6,
chercheur au Laboratoire d'informatique algorithmique : fondements et applications (LIAFA).