Les réseaux d'interaction

Yves Lafont
CNRS - Institut de Mathématiques de Luminy
163 avenue de Luminy, Case 907, 13288 Marseille Cedex 9
E-mail : `lafont@iml.univ-mrs.fr`

24 avril 1998

Modèles du calcul

Bien que le calcul soit pratiqué depuis des millénaires, les modèles abstraits et généraux du calcul ne sont apparus que dans les années 1930. Les plus connus sont les *machines de Turing*, qui préfigurent les ordinateurs, et le *lambda-calcul de Church*, qui a donné naissance à toute une famille de langages de programmation dits fonctionnels, tels que *LISP* et *ML* (voir les encadrés).

Il existe bien d'autres modèles, mais tous conduisent à la même notion de *fonction calculable*. La plupart des mathématiciens et des informaticiens s'accordent donc sur cette notion et considèrent que toute fonction calculable par un procédé mécanique quelconque est aussi calculable par une machine de Turing. C'est la *thèse de Church*, qui permet d'énoncer et de démontrer rigoureusement des résultats négatifs tels que l'indécidabilité du dixième problème de Hilbert : il n'existe aucun algorithme pour déterminer si une équation diophantienne quelconque admet une solution entière.

Les machines de Turing servent aussi à définir des notions plus fines qui expriment la faisabilité des calculs : fonctions calculables en temps polynômial, en espace polynômial, etc. Il n'en reste pas moins que ce modèle repose sur une conception totalement séquentielle du calcul, avec un processeur et une mémoire qui jouent des rôles bien différenciés.

A l'opposé de cette conception se trouve le modèle des *automates cellulaires*, qui repose sur le principe suivant : l'état d'une cellule à l'instant $n+1$ dépend uniquement de l'état de cette même cellule et de ses voisines à l'instant n. Il s'agit bien sûr d'un modèle de calcul parallèle, car toutes les cellules jouent des rôles analogues et contribuent collectivement à l'avancement du calcul. Notons que, pour le bon déroulement des opérations, il est indispensable que toutes les cellules changent d'état au même instant ; autrement dit, les transitions sont locales, mais synchrones. Cette temporalité globale n'est guère satisfaisante d'un point de vue strictement esthétique. Plus prosaïquement, elle impose de sévères limitations technologiques à la conception d'ordinateurs parallèles : nécessité d'une horloge commune, difficultés pour synchroniser des réseaux de machines distantes, etc.

Principes de l'interaction

Oublions provisoirement ces modèles, et essayons d'en concevoir un qui soit assez général, mais avec de bonnes propriétés. Tout d'abord, on peut voir un calcul comme une sorte de processus physique discret. Si l'espace de calcul est linéaire, comme dans le cas des machines de Turing, l'état du calcul à un instant donné sera représenté par un mot sur un alphabet fini. En toute généralité, ce sera un *réseau* formé de *cellules* étiquetées par un nombre fini de *symboles*. On suppose que toutes les cellules qui portent le même symbole ont le même nombre de *ports*, et que les ports des différentes cellules sont connectés deux par deux.

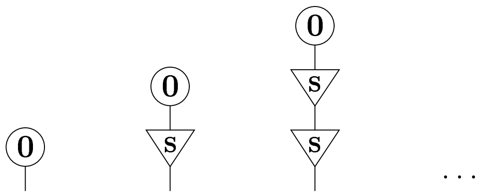
La dynamique du calcul s'exprime alors par un nombre fini de transitions locales. A priori, une telle transition est une règle de réécriture dont le membre gauche est un réseau connexe avec un certain nombre de *ports libres*, et le membre droit est un réseau quelconque avec le même nombre de ports libres. Pour que la transition soit vraiment locale, on se limite au cas où le membre gauche est formé de deux cellules connectées une seule fois. Pour éviter les conflits entre ces *règles d'interaction*, on fixe un *port principal* pour chaque cellule, les autres ports étant dits *auxiliaires*, et on exige que, dans le membre gauche de chaque règle, les deux cellules soient connectées par leurs ports principaux. Enfin, on admet au plus une règle par paire de symboles.

De la sorte, si deux règles s'appliquent à un même réseau, on peut les exécuter toutes les deux, l'une après l'autre ou simultanément, puisqu'elles portent nécessairement sur des sous-réseaux disjoints. Ici donc, nul besoin d'une temporalité globale : les transitions sont à la fois locales et asynchrones.

Travaux pratiques

Afin de bien comprendre comment tout cela fonctionne, essayons de programmer des opérations arithmétiques élémentaires. Si on n'a aucun souci d'efficacité, le plus simple est d'adopter la numération unaire : 0, s0 pour 1, ss0 pour 2, etc. On commence donc avec les deux symboles suivants :

D'une manière générale, on dessine une cellule comme un triangle avec le port principal sur l'un des sommets et les ports auxiliaires sur le côté opposé. Lorsqu'il n'y a qu'un port, celui-ci est forcément principal, et on dessine la cellule comme un rond. Les entiers naturels sont représentés par les réseaux suivants :



Un tel réseau est *irréductible* : tous les ports principaux sont connectés à des ports auxiliaires, sauf un qui est libre, et donc prêt à interagir avec son environnement.

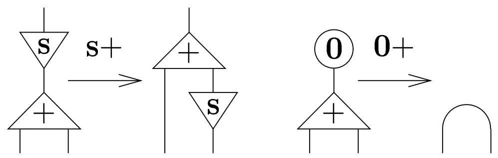
Pour programmer l'addition et la multiplication, considérons les équations qui les définissent :

$$\mathrm{s}x + y = \mathrm{s}(x + y), \qquad 0 + y = y,$$

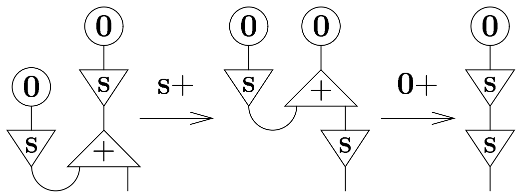
$$\mathrm{s}x \times y = (x \times y) + y, \qquad 0 \times y = 0.$$

Dans les deux cas, il s'agit d'une définition par récurrence sur le premier argument, et c'est donc là que doit se situer le port principal, les deux ports auxiliaires correspondant au second argument et au résultat de l'opération :

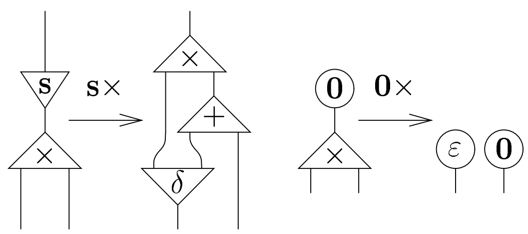
Pour l'addition, on obtient les deux règles suivantes :



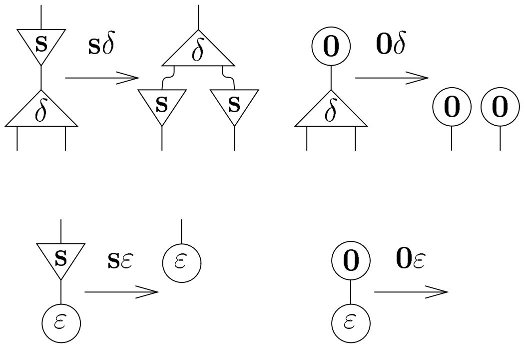
Et voici comment on calcule $1 + 1$:



Pour la multiplication, c'est un peu plus compliqué, car le second argument y est utilisé deux fois dans une règle, pas du tout dans l'autre. Il faut donc introduire un duplicateur et un effaceur :



Les mécanismes de duplication et d'effacement sont exprimés par les règles suivantes :



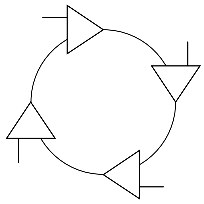
Le lecteur pourra s'amuser à calculer 2×2, en constatant que les règles peuvent souvent s'exécuter en parallèle.

Programmes et preuves

Il n'existe pas encore de véritable langage de programmation fondé sur cette notion d'interaction, mais on peut déjà entrevoir les avantages d'une telle approche en la comparant avec la programmation fonctionnelle, qui lui est apparentée. Tout d'abord, l'interprétation des règles est limpide : il n'y a aucune ambiguïté liée par exemple à la portée des variables ou à l'ordre d'exécution. De plus, la gestion de la mémoire est prise en compte par le programme lui-même, sous forme de règles d'effacement. Plus intéressant : ces réseaux représentent des processus interactifs, avec des entrées et des sorties entremêlées, et non pas seulement des fonctions. Enfin, un bon ordinateur parallèle devrait pouvoir tirer parti du caractère local et asynchrone des transitions.

A l'origine, les *réseaux d'interaction* n'ont pas été conçus comme un langage de programmation, ni comme un modèle général du calcul. Il s'agissait seulement de comprendre le mécanisme de réduction des démonstrations dans la *logique linéaire* de Jean-Yves Girard. Cette logique est un raffinement de la logique classique où l'on tient compte de la multiplicité de chaque hypothèse, ce qui permet de représenter les démonstrations sous une forme géométrique : les *réseaux de preuves*.

Pour être corrects, ces derniers doivent satisfaire une condition globale qui, dans sa formulation la plus simple due à Vincent Danos et Laurent Regnier, stipule que certains graphes dérivés du réseau lui-même sont connexes et acycliques. Or il se trouve que cette condition d'origine logique a une interprétation tout à fait intéressante dans le cadre des réseaux d'interaction. Elle empêche la formation de *cercles vicieux*, qui sont des configurations de blocage du type suivant :

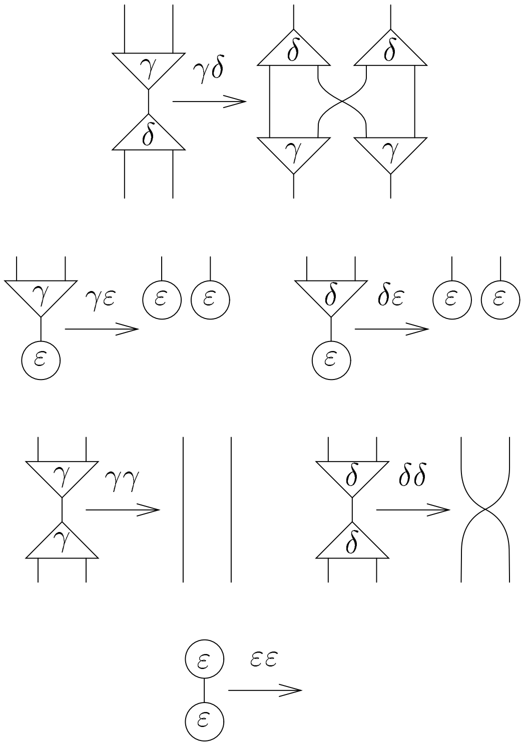


Etant donné que certaines règles, telles que $\mathbf{s}\times$ dans l'exemple ci-dessus, peuvent créer des cycles dans un réseau, il n'est pas du tout évident de prévoir la formation d'un tel cercle vicieux. En fait, c'est la condition d'acyclicité qui est utilisée ici, et on ne connaît pas de justification extra-logique pour la condition de connexité, mais c'est tout de même un bel exemple de transfert de concept entre la logique et l'informatique.

Combinateurs d'interaction

Nous avons déjà écrit un programme sous forme de règles d'interaction. On peut en imaginer bien d'autres, plus ou moins compliqués. Par exemple, il n'est pas difficile de simuler une machine de Turing, et même un automate cellulaire, avec des règles d'interaction. Une question se pose naturellement : existe-t-il un programme *universel*, c'est-à-dire dans lequel il est possible de traduire n'importe quel autre programme écrit avec des règles d'interaction ? Il faut se limiter à des traductions qui préservent la structure fine des calculs, car sinon, n'importe quelle machine de Turing universelle ferait l'affaire.

La réponse à cette question est positive, mais le plus étonnant est la simplicité de la solution. Il suffit en effet de trois *combinateurs* γ, δ et ε, avec les six règles suivantes :



On dit que les trois premières règles, qui font intervenir deux symboles distincts, sont des *commutations*, et les trois autres, qui font intervenir deux fois le même symbole, sont des *annihilations*. Ces règles rappellent celles qui ont été introduites par John Lamping dans un perspective très différente : il s'agissait de réaliser la *réduction partagée* du lambda-calcul.

La propriété d'universalité signifie que, pour un programme donné, il est possible d'associer à chaque symbole α à n ports un réseau de combinateurs $\Phi(\alpha)$ avec n ports libres, de telle façon que, pour chaque règle $R \to S$ du programme, le réseau $\Phi(R)$, c'est-à-dire R dans lequel on a remplacé chaque symbole par sa traduction, se réduit en $\Phi(S)$. Autrement dit, chaque règle du programme se traduit en une succession de règles d'interaction entre combinateurs. Ces dernières peuvent donc être considérées comme les règles élémentaires du calcul.

La démonstration de ce résultat non trivial s'inspire à la fois de la logique linéaire et de la biologie ! On commence en effet par associer au programme tout entier un réseau de combinateurs Γ, puis on transforme ce Γ en un réseau $!\Gamma$ qui ne contient aucun δ. Ce dernier est en quelque sorte le code génétique du programme : il peut être dupliqué, effacé, ou encore exécuté. La traduction d'un symbole contient essentiellement le numéro de ce symbole, ainsi que le code génétique, qui est commun à tous les symboles.

Bien que les réseaux de combinateurs ressemblent beaucoup aux réseaux *multiplicatifs*, qui sont les réseaux de preuve les plus simples, les liens avec la logique restent encore assez mystérieux.

Machines concrètes

Les réseaux d'interaction sont un modèle abstrait du calcul, moins abstrait certes, que le lambda-calcul, mais abstrait tout de même. Il semble en effet difficile d'imaginer un quelconque processus physique, chimique, ou même biologique, qui réaliserait *directement* les règles d'interaction de n'importe quel programme. La principale difficulté provient du fait que la géométrie du réseau évolue de façon imprévisible.

Il est donc tentant de rechercher d'autres modes d'exécution des réseaux, qui ne passent pas explicitement par le mécanisme de réduction. C'est dans ce but que Jean-Yves Girard a développé la *géométrie de l'interaction* pour les réseaux de preuves. L'idée consiste à interpréter un réseau comme un opérateur nilpotent sur un espace de Hilbert, et l'exécution comme un processus d'itération de cet opérateur. A vrai dire, la structure Hilbertienne n'est pas vraiment essentielle ici : l'important est d'avoir suffisamment de place.

Il se trouve que dans le cas des combinateurs d'interaction, la géométrie de l'interaction prend une forme particulièrement simple : elle fournit une correspondance bijective entre les réseaux de combinateurs et les *machines réversibles à 2 piles* sur un alphabet à 2 lettres. Ce sont des variantes des machines de Turing, où le ruban est remplacé par des piles. La réversibilité signifie qu'à partir de la configuration, c'est-à-dire de l'état et du contenu des piles, à un instant donné, il est toujours possible de retrouver la configuration précédente. Cette correspondance est très intéressante, mais pour l'instant, le mode d'exécution qui en découle n'est pas vraiment convaincant : non seulement, on perd le bénéfice du parallélisme, mais Vincent Danos et Laurent Regnier ont montré que l'exécution pouvait prendre un temps exponentiel dans des cas où la réduction est linéaire.

Un approche totalement différente consiste à se restreindre à *l'interaction dure*, où la géométrie des réseaux est invariante par réduction. Autrement dit, le membre droit de chaque règle d'interaction a la même forme que le membre gauche : seuls les symboles et les positions des deux ports principaux peuvent changer. C'est bien sûr une contrainte très forte, et d'ailleurs, aucune des règles considérées ci-dessus n'est de ce type. On imagine mieux comment réaliser physiquement ce genre d'interaction, mais il s'agit d'un langage de très bas niveau : toute la difficulté est alors de savoir comment compiler dans ce langage un programme tel que ceux qui sont considérés ci-dessus.

Les machines de Turing

Une machine de Turing est une sorte de calculatrice stylisée. On ne cherche pas à décrire le fonctionnement du processeur ; on sait seulement qu'il peut prendre un nombre fini d'états. La mémoire, par contre, doit être potentiellement infinie : c'est un ruban illimité avec des cases qui contiennent chacune une lettre d'un alphabet fini. A un instant donné, la tête de lecture permet d'accéder à une seule case de la mémoire.

Le comportement de la machine est complètement déterminé par sa *table de transition*, qui à un état q et à une lettre a associe l'instruction qui doit être exécutée lorsqu'on lit a dans l'état q. Hormis l'arrêt, toutes les instructions sont de la forme suivante : écrire a' (à la place de a), déplacer la tête de lecture vers la gauche (ou vers la droite), et passer dans l'état q'.

Une telle table peut être considérée comme un programme écrit dans un langage de programmation de très bas niveau. Bien que ce soit un peu fastidieux, on peut ainsi construire la table d'une machine de Turing *universelle* capable de simuler n'importe quelle autre machine de Turing, de même qu'un ordinateur est capable de simuler n'importe quelle calculatrice.

Le lambda-calcul

Le lambda-calcul est un système de notation pour les fonctions. Si U est un terme, $\lambda x.U$ représente la fonction qui à x associe U (en général, U contient une ou plusieurs occurrences de la variable x). Si U et V sont des termes, $U\,V$ représente l'application de la fonction U à l'argument V. Il est commode d'écrire $U\,V\,W$ pour $(U\,V)\,W$: cela permet de voir U comme une fonction à 2 arguments.

Une seule règle de calcul suffit : le terme $(\lambda x.U)\,V$ se réduit en $U[V/x]$, c'est-à-dire U dans lequel on a remplacé les occurrences de la variable x par l'argument V. En fait, le mécanisme de substitution est un peu plus subtil : on ne doit remplacer que les occurrences de x qui sont *libres* dans U. Dans sa version *non typée*, le lambda-calcul est très puissant car il permet d'appliquer n'importe quelle fonction à n'importe quel argument. D'ailleurs, il arrive que le calcul ne termine pas.

Il n'y a pas de notation spécifique pour les arguments primitifs, mais on peut représenter les entiers naturels par les fonctions $\underline{0} = \lambda f.\lambda x.x$, $\underline{1} = \lambda f.\lambda x.f\,x$, $\underline{2} = \lambda f.\lambda x.f\,(f\,x)$, etc. L'addition et la multiplication sont alors respectivement représentées par les termes $A = \lambda p.\lambda q.\lambda f.\lambda x.p\,f\,(q\,f\,x)$ et $M = \lambda p.\lambda q.\lambda f.p\,(q\,f)$. Par exemple, $A\,\underline{2}\,\underline{3}$ se réduit en $\underline{5}$, et $M\,\underline{2}\,\underline{3}$ se réduit en $\underline{6}$. La thèse de Church affirme que toute fonction *calculable* $\varphi : \mathbf{N}^k \to \mathbf{N}$ est représentée par un terme du lambda-calcul.

Exercice : quelle opération est représentée par le terme $\lambda p.\lambda q.p\,q$?

Pour en savoir plus ...

- sur la *logique linéaire* : J.-Y. Girard, « Linear logic : its syntax and semantics », in « Advances in Linear Logic », London Mathematical Society Lecture Note Series 222, p. 1–42, Cambridge University Press, 1995 ;
- sur les *réseaux de preuve* : Y. Lafont, « From Proof-Nets to Interaction Nets », idem, p. 225–247 ;
- sur la *géométrie de l'interaction* : V. Danos & L. Regnier, « Proof-Nets and the Hilbert space », idem, p. 307–328 ;
- sur les *réseaux d'interaction* et les *combinateurs d'interaction* : Y. Lafont, « Interaction Combinators », Information and Computation 137 (1), p. 69–101, Academic Press, 1997.