par L.T.D. Nguyễn et J. Ledent

Une preuve de maths est un programme informatique !

À la fin des années 1960 émerge l'idée qu'une preuve mathématique correspond à un programme informatique. On rêve ainsi de comprendre les programmes sous-jacents à toutes les démonstrations mathématiques ! La « correspondance preuves–programmes » permet de concevoir des langages de programmation où les programmes sont mathématiquement prouvés sans *bugs*.

Au tournant des XIXe et XXe siècles, la théorie de la démonstration naissante formalise les notions de proposition et de preuve, ce qui permet d'obtenir des résultats métamathématiques remarquables, comme les théorèmes d'incomplétude de Gödel en 1931 (voir dans un précédent dossier). Peu après, en 1936, Gerhard Gentzen fournit une preuve de la consistance de l'arithmétique qui emploie des méthodes inédites. Une fois éclairées par les concepts de l'informatique, ces méthodes seront essentielles pour ouvrir une nouvelle ère de la logique, avec de nombreuses applications : vérification des démonstrations, écriture de programmes certifiés sans *bugs*, et même remise en cause des fondements des mathématiques.
La preuve de consistance de Gentzen contredit-elle le théorème de Gödel ? Non, car elle fait appel à un principe logique plus fort que l'arithmétique de Peano, qui est incapable de démontrer seule sa propre consistance. Ce principe intervient simplement pour montrer que l'algorithme d'élimination des coupures ne fait pas de boucle infinie (voir plus loin), ce qui n'est décidément pas une chose triviale !

Une structure cachée dans les preuves

Le point de départ de notre histoire, c'est une tentative de modéliser fidèlement les raisonnements mathématiques courants. Par exemple, à partir d'une hypothèse « P ou Q », on peut raisonner par disjonction de cas, alors que si l'on veut conclure une telle proposition, on va chercher soit à établir P, soit à établir Q. Ainsi, pour chaque connecteur logique, on sera amené à définir des règles d'introduction et d'élimination, qui décrivent respectivement comment on prouve et utilise les formules commençant par ce connecteur.

Un autre exemple fondamental, c'est la règle d'élimination de l'implication, appelée traditionnellement *modus ponens*. Cette règle énonce que si l'on sait que P est vrai et P ⇒ Q est vrai, alors on peut en déduire Q. En enchaînant de telles règles dans des arbres (voir en encadré), on construit des démonstrations. Ce qui fournit une définition de ce qu'est une preuve formelle. Dans l'idée, une argumentation écrite en langue naturelle, sur papier, devrait pouvoir être traduite en un arbre de preuve en la décomposant en ses étapes élémentaires. Dans ce formalisme, que Gentzen baptise *déduction naturelle*, on peut aussi définir des règles pour gérer les quantificateurs « pour tout » (∀) et « il existe » (∃), puis des règles qui expriment les axiomes de l'arithmétique… : essentiellement, il devient possible de capturer divers modes de raisonnement mathématique.

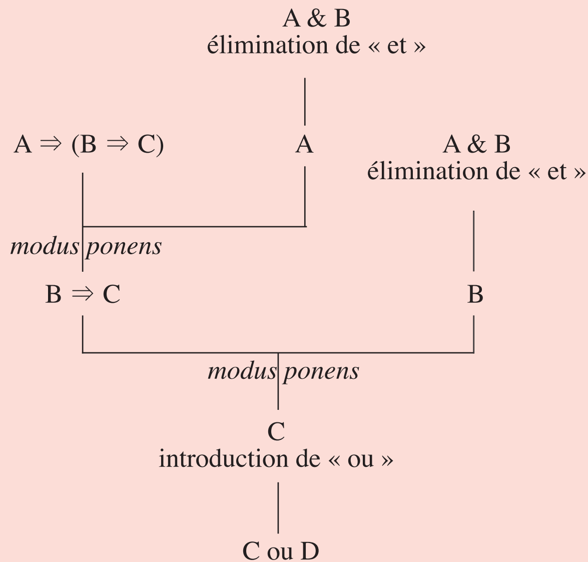
Le théorème le plus profond démontré par Gentzen est relatif à l'élimination des coupures. De quoi s'agit-il ?
Dans une preuve informelle, on « effectue une coupure » à chaque fois que l'on prouve un lemme intermédiaire pour s'en resservir par la suite, que l'on introduit une variable symbolique qui sera fixée plus tard… bref, dès que l'on prouve quelque chose de façon indirecte.
En déduction naturelle, une coupure se produit lorsqu'une règle d'introduction et une règle d'élimination correspondante se suivent dans l'arbre de preuve.
Éliminer les coupures, c'est réécrire la preuve pour se débarrasser de ce caractère indirect. Ce qui signifiera, par exemple, que l'on va remplacer une preuve de Q, obtenue par *modus ponens* à partir d'une preuve (notée π_1) de P et d'une preuve (notée π_2) de P ⇒ Q, par une preuve plus « directe » de Q,

La déduction naturelle

Les règles en déduction naturelle suivent toutes le même schéma : elles prennent en hypothèse une ou plusieurs propositions, en en déduisent une unique conclusion. On peut représenter cela graphiquement par un nœud conclusion relié à des nœuds hypothèses. Voici par exemple comment s'écrit l'élimination de l'implication :

$$\frac{A \Rightarrow B \qquad A}{A}$$ *(modus ponens).*

Pour écrire l'introduction de l'implication, il faut légèrement complexifier le format pour tenir compte des hypothèses dans le contexte. Informellement, la règle dit que prouver A ⇒ B, c'est supposer A puis prouver B.
Construire une preuve, c'est relier les nœuds par les liens des règles pour obtenir un arbre, qui démontre la conclusion à la racine à partir des hypothèses aux feuilles. Voici un exemple d'arbre de preuve (aujourd'hui, on utilise un formalisme différent, voir en page suivante) :



Cette déduction prouve C ou D à partir des hypothèses A ⇒ (B ⇒ C) et A & B (qui apparaît deux fois en feuille de l'arbre). Elle utilise les règles d'élimination de & (« et »), celles de ⇒ (« implique ») et l'introduction de « ou ».

consistant en la preuve π_2 où le lemme P est redémontré par π_1 à chaque fois qu'il est utilisé. La nouvelle preuve obtenue est susceptible de contenir encore des coupures, que l'on peut à nouveau éliminer, et ainsi de suite.
Le théorème de Gentzen affirme simplement que cette procédure débouche en fin de compte, au bout d'un nombre fini d'étapes, sur une preuve sans coupures de Q. Or une telle preuve possède une structure tellement simple qu'il est facile de voir qu'elle ne peut pas mener à une contradiction : c'est de cette manière que l'on démontre la consistance de l'arithmétique, en la réduisant à une question de terminaison algorithmique. L'informatique fait irruption en logique…

Correspondance entre preuves et programmes

Faisons un bond en avant vers la fin des années 1960. Le logicien américain William Alvin Howard (né en 1926), s'inspirant d'une remarque de son homologue Haskell Brooks Curry (1900–1982), découvre un lien profond entre la déduction naturelle et le lambda-calcul, un langage de programmation théorique étudié par les informaticiens comme modèle de calcul universel (on peut y coder toute fonction calculable par un ordinateur). C'est notamment le fondement mathématique du langage Caml, développé par l'Inria en 1985 et enseigné en classes préparatoires et dans certains cursus universitaires.

Au départ, le lambda-calcul était très minimaliste : des variables, des fonctions, et c'est tout. Puis on a voulu ajouter des fonctionnalités pour modéliser les langages de programmation réels. Parmi celles-ci, on trouve des mécanismes de classification des données manipulées, qui permettent par exemple de ne pas utiliser accidentellement un entier là où il faudrait un vecteur. De tels mécanismes s'appellent des *systèmes de types*. Un type est comparable à une unité en physique (mètre, seconde…), et, de même que l'on attend des formules physiques qu'elles n'additionnent pas des longueurs et des durées, on souhaite que les programmes que l'on écrit soient *bien typés*, pour éviter certains bugs.
La remarquable correspondance de Curry–Howard affirme que les preuves en déduction naturelle correspondent parfaitement aux programmes du lambda-calcul, annotés avec des informations de typage !
Avec un dictionnaire entre formules et types, « P et Q » devient le type des couples de P et de Q, c'est-à-dire P × Q. De même, « P ⇒ Q » devient le type des fonctions de P dans Q… On établit une correspondance entre règles de déduction et instructions de programmation : ainsi, le *modus ponens* devient l'application d'une fonction de type P ⇒ Q à un argument de type P pour obtenir un résultat de type Q.

Élimination des coupures pour la conjonction

Voici un exemple de coupure : on dispose d'une preuve π_P de P et d'une preuve π_Q de Q. On déduit d'abord P & Q avec la règle d'introduction de &, puis P par la règle d'élimination du même &. Cette coupure s'élimine : on prouve P directement avec π_P, et on « jette » π_Q. Dans un formalisme moderne :

$$\dfrac{\dfrac{\dfrac{\pi_P}{P} \quad \dfrac{\pi_Q}{Q}}{P \,\&\, Q}}{P} \to \dfrac{\pi_P}{P}$$

Que calculent ces programmes ? En règle générale, le type du programme nous informe sur son résultat : un programme de type τ va produire une valeur de type τ. Pour le type des entiers, c'est simple : on veut que les valeurs ayant ce type soient des nombres entiers. De façon analogue, une preuve d'une formule P va calculer une « valeur de type P ». Surprise : une valeur de type P, n'est qu'une preuve sans coupures de P ! En fait, l'algorithme d'élimination des coupures exécute les programmes en lambda-calcul, de la même manière que votre navigateur Web exécute du JavaScript : c'est un algorithme universel. Parmi tous les programmes possibles, qui peuvent « planter » ou contenir une boucle infinie, le typage permet de sélectionner ceux qui se comportent bien : c'est pour cela, fondamentalement, que l'élimination des coupures termine.

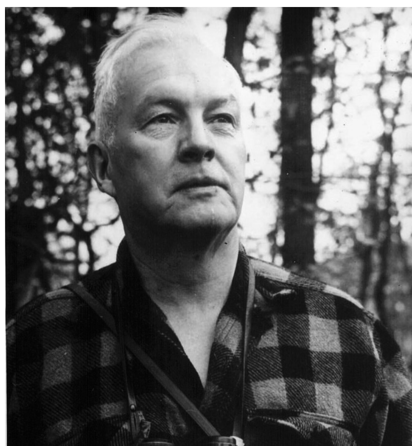
Pour résumer :

- formule logique = type ;
- élimination des coupures = exécution d'un programme ;
- preuve correcte = programme sans bug (typé) !

L'intuitionnisme, logique du calcul

Concrètement, si l'on regarde comment s'exécute une preuve π de P = « il existe x tel que $Q(x)$ », elle donne une preuve sans coupures de P qui, forcément, démontre d'abord $Q(t)$ pour un certain terme t puis en déduit P (c'est la règle d'introduction de « il existe »). Ainsi, π n'est pas juste une preuve d'existence d'un tel x, c'est aussi un programme qui permet de trouver un x convenable.

Pourtant, dans les mathématiques usuelles, il est courant d'affirmer l'existence d'objets avec des arguments non



Haskell Brooks Curry.

constructifs, sans fournir de méthode pour les obtenir (voir en page 6). La correspondance preuves–programmes serait-elle une recette miracle pour trouver de telles méthodes ? Pas tout à fait : la déduction naturelle formalise en fait non pas la logique classique, mais la logique intuitionniste, plus restrictive, où toute démonstration est automatiquement constructive.

Le prix à payer pour la constructivité, c'est la perte du principe du tiers exclu. Son énoncé est simple : pour toute proposition P, ou bien P est vraie ou sa négation est vraie. Ça tombe sous le sens, mais le problème, c'est que si l'on veut affirmer constructivement « P ou (non P) », il faut être capable de décider si c'est P qui est vrai, ou si c'est non P : on peut avoir besoin de savoir dans quel cas on est lors d'une construction qui dépend de cette disjonction. Rien à voir avec la logique classique, pour laquelle toute formule prend une valeur de vérité (vrai ou faux), indépendamment de nos moyens de le savoir.

Réciproquement, ajouter l'axiome du tiers exclu (ou celui, équivalent, d'*élimination de la double négation* : « non (non P) implique P ») à la logique intuitionniste nous fait retomber dans la logique classique.

Curry–Howard à la conquête du monde

Faut-il alors se résoudre à ce que les maths « ordinaires » soient à jamais hors de portée de la correspondance preuves–programmes, de par leur non-constructivité essentielle ? Eh bien non ! Au début des années 1990, Timothy George Griffin découvre que l'élimination de la double négation peut être interprétée par une instruction sophistiquée de programmation. Depuis, les logiciens ont compris que la logique classique décrit des programmes interagissant avec leur environnement : dans un vrai système informatique, un programme ne tourne jamais seul, il doit discuter avec le système d'exploitation, les périphériques…
Ces idées sont poussées très loin dans le programme de recherche récent de la *réalisabilité classique* (porté par Jean-Louis Krivine, voir *la Logique*, Bibliothèque Tangente 15, pages 80 à 95, 2004), qui cherche à interpréter les axiomes de la théorie des ensembles, sur lesquels se basent les mathématiques, par des instructions de « programmation bas niveau ». L'espoir ultime est de pouvoir comprendre les programmes sous-jacents à toutes les démonstrations des « vraies » mathématiques…

Et pourquoi ne pas faire encore plus radical et chercher à supplanter la vieille théorie des ensembles, en fondant les mathématiques directement sur le paradigme « propositions comme types » ? C'est ce qu'accomplit la *théorie des types*, qui sert notamment de base à Coq, un logiciel développé par des chercheurs français qui permet de formaliser sur ordinateur et vérifier automatiquement des démonstrations mathématiques (voir *Maths et Informatique*, Bibliothèque Tangente 52, 2014). Coq permet de s'assurer que des enchaînements sophistiqués d'arguments ne contiennent pas d'erreur, ce qui a été fait en 2012 pour le théorème de Feit–Thompson, résultat de théorie des groupes dont la démonstration s'étale sur deux livres (voir *les Maths de l'impossible*, Bibliothèque Tangente 49, 2013). Pour cela, il a fallu formuler les concepts et théorèmes classiques d'algèbre au sein de la théorie des types, ce qui montre bien que l'on peut faire de vraies maths sans théorie des ensembles !

En outre, grâce à l'informatique, les bénéfices de la logique intuitionniste se concrétisent : Coq permet d'exécuter une preuve constructive pour calculer un témoin d'existence. Donc non seulement c'est un système de preuves mathématiques, mais il permet aussi d'écrire des programmes, et même de prouver qu'ils font bien ce que l'on attend d'eux. Ainsi, le compilateur CompCert, développé à l'INRIA, est certifié sans *bug* ! Plus généralement, la correspondance preuves–programmes sert d'inspiration à la conception des langages de programmation fonctionnels, de plus en plus utilisés dans l'industrie.

Au-delà des applications informatiques, la théorie des types offre des connexions inattendues avec la topologie algébrique et la théorie des catégories, qui ouvrent des perspectives pour un fondement nouveau des mathématiques : la théorie homotopique des types, qui fait l'objet d'un article dans ce même dossier…

L.T.D. N. & J.L.

La logique linéaire

En plus d'expliquer le contenu calculatoire des logiques préexistantes (classique, intuitionniste...), la correspondance de Curry–Howard a aussi inspiré la création de la *logique linéaire* de Jean-Yves Girard. Cette dernière considère les formules comme des « ressources » : utiliser une hypothèse la consomme et on ne peut plus s'en resservir. Cela conduit à faire une distinction entre deux « et » différents (de même pour deux « ou » différents). Ainsi, « A et B » peut correspondre : soit à $A \otimes B$ (« je dispose simultanément d'un A et d'un B »), soit à $A \,\&\, B$ (« je dispose d'une ressource qui peut être convertie en un A et peut être convertie en un B, mais je dois choisir entre les deux »).
En notant $\multimap$ l'implication linéaire, on peut se convaincre que l'on a $A \multimap A \,\&\, A$, mais pas $A \multimap A \otimes A$ (cette dernière formule signifierait que l'on a deux A pour le prix d'un !).

Au restaurant, ça marche !

Une application classique est la description des menus de restaurant :
6,50 € $\multimap$ Entrée $\otimes$ Plat $\otimes$ Dessert, où Dessert = Beignet pomme & Litchi.
Tout ceci est naturel avec un point de vue procédural ; ça l'est beaucoup moins si l'on se dit qu'une proposition vraie le reste si l'on s'en sert ! Pour incorporer la pérennité de la vérité à la logique linéaire, on introduit l'*exponentielle* !A (la ressource !A fournit autant de copies de A que l'on pourrait souhaiter avoir). Ce qui permet de retrouver la logique classique : ainsi, l'implication usuelle $A \Rightarrow B$ s'exprime avec l'implication linéaire en écrivant $!A \multimap B$.

Les exponentielles, ressources inépuisables, sont une manifestation de l'infini potentiel en logique. « Potentiel » parce que !A n'est qu'un processus permettant de produire toujours plus de A sans que leur quantité cesse d'être finie, contrairement à l'infini actuel des ensembles qui contiennent vraiment une quantité infinie d'éléments.
La théorie des ensembles a permis d'établir une hiérarchie des infinis actuels : on sait ainsi qu'il y a « autant » de nombres entiers que de nombres rationnels, alors que les nombres réels sont « strictement plus » nombreux. La logique linéaire ouvre la voie vers une théorie semblable des infinis potentiels.
Son penchant informatique serait alors la théorie de la complexité, qui classifie les problèmes et algorithmes en fonction de leur temps de calcul. Peut-on espérer une réponse au problème P *versus* NP à l'aide de la logique ? Le domaine de recherche de la *complexité implicite* s'y attaque...