Du programme de Hilbert aux programmes informatiques

Jean-Louis Krivine

Université Paris-Diderot, C.N.R.S.

Bordeaux, 19 janvier 2012

Le programme de Hilbert

On va parler de logique, plus précisément de *théorie des démonstrations*. C'est un domaine considéré en général comme inutile par les autres mathématiciens qui disent : "On n'a pas besoin de vous, les logiciens, pour savoir faire des démonstrations correctes. L'important, en mathématiques se trouve dans la compréhension profonde du sujet, et pas dans la formalisation". J'espère vous convaincre qu'il s'agit, en fait, d'un sujet formidablement intéressant à tous points de vue : intérêt mathématique, philosophique et pour les applications.

On va d'abord faire un peu d'histoire de la logique au XXème siècle. Il y a une bande dessinée là-dessus, qui est fort bien faite et qui raconte les aventures de Bertrand Russell ; elle s'appelle Logicomix. Elle est assez approximative du point de vue historique, mais je le serai aussi. Par contre, j'irai beaucoup plus loin dans le temps, jusqu'à aujourd'hui, et beaucoup plus vite.

La logique est née d'une volonté de comprendre la nature du *raisonnement mathématique* et la raison de son efficacité surprenante ("déraisonnable" comme dit le physicien Eugène Wigner).

La première chose à faire était de l'expliciter complètement, de donner les règles du jeu. C'est ce qui a été terminé dans les années 1930, avec :

• *Le théorème de complétude* (Gödel,1930) qui donne les règles de déduction de la "logique pure" (qu'on appelle le "calcul des prédicats").
Le théorème de complétude est remarquable, parce qu'il donne la liste des règles de déduction, et qu'il montre qu'on ne peut rien leur ajouter, elles contiennent tous les raisonnements possibles en mathématiques. Ce qui est étonnant, c'est qu'on puisse *démontrer* une chose pareille. Pourtant, une fois qu'on a vu la preuve, on est convaincu.

• *Les axiomes de la théorie des ensembles* dite de Zermelo-Frænkel (Zermelo 1908 ; Frænkel et Skolem 1922). C'est à ces axiomes qu'on applique ces règles de déduction.

Certains ont alors cru que le travail était fini et que les mathématiques n'étaient pas autre chose que le jeu formel d'application des règles de déduction à partir des axiomes.
Sans aller jusque-là, il était trop malin pour ça, Hilbert a vu dans la formalisation des mathématiques, un moyen de garantir l'utilisation de notions abstraites et plus ou moins ésotériques, qui risquaient d'aboutir à une contradiction : les infinis de taille variée, l'axiome du

choix, etc. On en a depuis introduit bien d'autres, comme les grands cardinaux (qui sont vraiment très grands) ou l'axiome de détermination.
Il faut bien avouer que le recours à "l'intuition" pour justifier tous ces axiomes n'est vraiment pas convaincant ; par exemple, l'axiome du choix et ses conséquences très bizarres, ou même l'existence de diverses tailles d'infini, qui n'a absolument rien d'intuitif.
A l'époque de Hilbert, certains de ces axiomes aboutissaient à des contradictions, le plus célèbre étant l'axiome de compréhension de Frege, contredit par le paradoxe de Russell.
Le *programme de Hilbert* consistait donc à démontrer la non contradiction de la théorie des ensembles, et pour commencer, celle de l'arithmétique, par des moyens dits "finitistes", autrement dit au-delà de tout soupçon (pas le droit, bien sûr, d'utiliser un ensemble infini).
Après quoi, nous pourrions revenir à notre activité mathématique habituelle, en nous servant tranquillement d'ensembles infinis de taille quelconque, de l'axiome du choix et autres abstractions, avec la garantie qu'il n'y aurait pas de catastrophe.

Il y a, dans ce programme, l'idée très intéressante et très juste, que derrière les notions et axiomes très abstraits, se cache une manipulation concrète de structures finies. Mais quels sont, au juste ces objets finis ?

Le théorème d'incomplétude de Gödel

Hilbert a cru qu'il s'agissait des formules mathématiques avec leur syntaxe et leurs règles de démonstration. C'était très naturel, mais c'est pourtant là-dessus qu'il s'est trompé ; on ne peut pas le lui reprocher, car ces mystérieux objets n'existaient pas encore, à cette époque (plus exactement, ils étaient en train d'être inventés, mais on était bien loin de faire le rapprochement).
Le défaut essentiel du programme de Hilbert est qu'il ne cherche pas à *expliquer* l'activité mathématique, mais seulement à la *sécuriser*. Il sait, et nous le savons aussi, que c'est une activité fort intéressante ; il veut pouvoir continuer à s'y consacrer, sans chercher à comprendre pourquoi. Mais c'était bien le maximum qu'on pouvait alors espérer.

Il est alors arrivé un évènement étonnant : l'erreur de Hilbert, l'impossibilité de mener à bien ce programme, a été *démontrée* et cela très rapidement (1931). C'est le *théorème d'incomplétude* de Gödel (qui, bien entendu, ne contredit pas son théorème de complétude) :

Si une théorie axiomatique $\mathscr{T}$ contient l'arithmétique de Peano et si $\mathscr{T} \vdash Cons(\mathscr{T})$, alors $\mathscr{T}$ est contradictoire.

La notation $\mathscr{T} \vdash F$ signifie qu'on peut déduire la formule F à partir des axiomes de $\mathscr{T}$; Cons($\mathscr{T}$) est la *formule* : "la théorie $\mathscr{T}$ est consistante", autrement dit "on ne peut pas déduire $\bot$ en appliquant les règles de démonstration à partir des axiomes de $\mathscr{T}$" ;
$\bot$ désigne une formule identiquement fausse (par exemple $0 \neq 0$).
Remarque. Attention, Cons($\mathscr{T}$) est bien une *formule d'arithmétique* ; cela a donc un sens de se demander si elle est, ou non, conséquence de la théorie $\mathscr{T}$.

Il s'agit du théorème à la fois le plus célèbre et le plus mal compris qui soit. Il est vrai qu'il faut vraiment y réfléchir à deux fois pour le comprendre, car il est violemment contraire à l'intuition. Voici deux remarques fort éclairantes à ce sujet :

• Pour montrer que $\mathscr{T} \vdash F$, tout le monde sait qu'on peut ajouter à $\mathscr{T}$ l'hypothèse $\neg F$: c'est le "raisonnement par l'absurde". Il est très utile car on a droit ainsi, gratuitement, à

une hypothèse supplémentaire. Bien entendu, on n'a quand même pas le droit d'ajouter l'hypothèse F elle-même, ça deviendrait trop facile !
Cependant, et c'est beaucoup moins connu, le fait est qu'on peut ajouter aussi l'hypothèse : "F est démontrable dans la théorie $\mathscr{T}$". C'est tout à fait contre-intuitif, on a l'impression qu'on suppose que le travail est déjà fait. C'est pourtant exactement ce que dit le théorème de Gödel.

• Autre remarque : on énonce souvent le théorème de Gödel de façon erronée, en disant que la formule $\mathrm{Cons}(\mathscr{T})$ est *indécidable* dans $\mathscr{T}$ (c'est-à-dire qu'on ne peut démontrer dans $\mathscr{T}$ ni cette formule, ni sa négation). Cet énoncé est *faux*, $\mathrm{Cons}(\mathscr{T})$ est seulement *indémontrable* (c'est-à-dire qu'on ne peut pas la démontrer dans $\mathscr{T}$), et ce n'est pas du tout pareil.
Autrement dit, *une théorie non contradictoire peut très bien montrer sa propre contradiction*. Par exemple $\mathrm{PA} + \neg\mathrm{Cons}(\mathrm{PA})$ (où PA désigne l'arithmétique de Peano).

Le théorème de Gödel nous apprend donc que le programme de Hilbert ne peut être basé sur les formules, leur syntaxe et leurs règles de démonstration. On en a conclu, un peu vite, qu'il fallait l'abandonner complètement et que les mathématiques devaient parler d'autres chose que de structures finies. Mais alors, de quoi parlent-elles donc ? Cela restait un mystère et cela permit de développer des idées fumeuses sur la réalité des objets mathématiques et la caverne de Platon. Les objets mathématiques existeraient réellement, mais dans un monde idéal. On s'approche du sexe des anges et de la pierre philosophale.

La correspondance preuves-programmes en logique intuitionniste

Or, au même moment (1930) à Göttingen, Haskell Curry, un étudiant en thèse de P. Bernays et D. Hilbert (encore lui), développait une théorie formelle de la substitution, qu'il appelait la *logique combinatoire*. Quelques années après (1936), Alonzo Church inventait une structure algébrique très voisine, le *lambda-calcul*. C'est une sorte d'algèbre avec deux opérations appelées *application* et *abstraction*. La première est tout à fait anodine, c'est une opération binaire, sans axiome. La deuxième est tout à fait nouvelle, elle s'applique à une variable et elle la rend muette ; on la note λx. C'est encore une théorie de la substitution, car la règle fondamentale du λ-calcul, appelée β-réduction, fait passer de $(\lambda x t)u$ à $t[u/x]$.
Le λ-calcul permit à A. Church de définir la notion de "programme de calcul" et de démontrer qu'aucun programme ne pouvait décider de la vérité d'un énoncé arithmétique.

Ces deux théories devaient rester très marginales et confidentielles, jusqu'en 1958, où fut inventé, par John Mc Carthy, le deuxième langage de programmation, après le FORTRAN ; c'est le LISP, qui est basé sur le λ-calcul.
A partir de là, la logique combinatoire et le lambda-calcul prirent une très grande importance, non pas encore en logique, mais en informatique (théorie des langages de programmation). Et, comme vous le savez, l'informatique commençait alors une ascension vertigineuse qui n'est pas près de se terminer.

C'est Curry qui s'aperçut le premier, en 1958, de l'analogie formelle entre la logique combinatoire et les règles de démonstration du système de déduction de Hilbert (toujours lui) en logique propositionnelle *intuitionniste*.
Puis, en 1969, William Howard observe la même analogie entre le lambda-calcul et les règles de la *déduction naturelle intuitionniste* de Gentzen. La *correspondance de Curry-Howard*, ap-

pelée aussi *correspondance preuves-programmes*, venait de faire son apparition.
C'est elle qui va nous occuper aujourd'hui ; elle est devenue un domaine de recherche fascinant qui touche à :

• L'informatique : c'est, en effet, la voie de recherche la plus intéressante dans le domaine des *preuves de programme*. Un problème capital pour l'informatique, vu la taille et la complexité des programmes utilisés, surtout dans les systèmes d'exploitation et les réseaux, est l'écriture de programmes *logiquement corrects*. On peut comprendre facilement qu'une correspondance entre les preuves et les programmes soit un outil précieux pour cela.

• La logique : cette correspondance introduit en logique, particulièrement en théorie des démonstrations et en théorie des ensembles, des outils intuitifs nouveaux et très puissants, qui sont les *programmes*.

• La philosophie des mathématiques : les objets finis qui se cachent derrière les abstractions mathématiques ne sont pas les démonstrations, mais *les programmes*. C'est ce changement de point de vue qui remet en selle le programme de Hilbert, avec un but beaucoup plus ambitieux : il ne s'agit plus de "sécuriser" l'activité mathématique, mais de comprendre sa nature et son objectif.

L'importance de la correspondance de Curry-Howard n'est pas été perçue immédiatement, loin de là. En effet, à ses débuts, elle ne concerne qu'un système ridiculement faible et peu expressif : la logique propositionnelle intuitionniste.
Vous prenez des variables propositionnelles $p, q, r, \ldots$ dont l'une est notée $\bot$
et le seul connecteur $\to$.
Les règles de la *déduction naturelle* sont :

1. $A_1, \ldots, A_n \vdash A_i$ (pour $1 \leq i \leq n$).
2. $A_1, \ldots, A_n \vdash A \to B$, $A_1, \ldots, A_n \vdash A \;\Rightarrow\; A_1, \ldots, A_n \vdash B$ (modus ponens).
3. $A_1, \ldots, A_n, A \vdash B \;\Rightarrow\; A_1, \ldots, A_n \vdash A \to B$.
4. $A_1, \ldots, A_n, \bot \vdash A$.

Exercice. Prouver $\vdash A \to A$; $\vdash (A \to (B \to C)), (A \to B), A \to C$;
on utilise la notation $A, B, C \to D$ pour $(A \to (B \to (C \to D)))$.

On ne va pas pouvoir exprimer grand'chose dans un tel système, car on est en *logique propositionnelle*. Cette limitation sera vite dépassée, mais il y en a une autre qui paraît insurmontable : on ne peut pas montrer toutes les formules valides, par exemple la *loi de Peirce* $((A \to B) \to A) \to A$; en effet, on est en *logique intuitionniste*.

Dans ces conditions, la correspondance preuve-programme apparaît d'abord comme une remarque intéressante, sans plus. En quoi consiste-t-elle ?
On "décore" les preuves avec des *programmes* qui sont des termes du λ-calcul :

1. $x_1 : A_1, \ldots, x_n : A_n \vdash x_i : A_i$.
2. $x_1 : A_1, \ldots, x_n : A_n \vdash t : A \to B$, $x_1 : A_1, \ldots, x_n : A_n \vdash u : A \;\Rightarrow\; x_1 : A_1, \ldots, x_n : A_n \vdash (t)u : B$.
3. $x_1 : A_1, \ldots, x_n : A_n, x : A \vdash t : B \;\Rightarrow\; x_1 : A_1, \ldots, x_n : A_n \vdash \lambda x\, t : A \to B$.
4. $x_1 : A_1, \ldots, x_n : A_n, x : \bot \vdash x : A$.

Dans l'expression $t : A$, qui se lit " t est de type A ", A est une formule propositionnelle et t est un *λ-terme* ; la définition des λ-termes est très simple, on les construit, à partir d'une liste de variables $x, y, \ldots$ de la façon suivante :
Chaque variable est un λ-terme.

Si t, u sont des λ-termes, $(t)u$ est un λ-terme ; cette opération est l'*application* ;
$(\ldots(((t)u_1)u_2)\ldots)u_n$ sera souvent noté $(t)u_1\ldots u_n$ ou même $tu_1\ldots u_n$.
Si t est un λ-terme et x une variable, alors $\lambda x\, t$ est un λ-terme ; cette opération est appelée *abstraction*.

Par exemple $\lambda x\, y\,, (\lambda x(x)x)\lambda x(x)x$ sont des λ-termes.

Le λ-calcul est l'archétype des langages de programmation. L'exécution d'un programme écrit dans ce langage se fait au moyen de la règle suivante appelée β-réduction :

$$(\lambda x\, t)u \rightsquigarrow t[u/x]$$

Un sous-terme de la forme $(\lambda x\, t)u$ s'appelle un *redex*. Il faut fixer une stratégie de réduction, par exemple la *réduction gauche* : on réduit toujours le redex le plus à gauche.

C'est ainsi que chaque preuve en logique propositionnelle intuitionniste donne un λ-terme et donc un programme.
Par exemple, on obtient ainsi :
$\vdash \lambda x\, x : A \to A$; $\vdash \lambda x\, x : \bot \to A$.
$\vdash \lambda x\lambda y\lambda z(xz)(y)z : (A \to (B \to C)), (A \to B), A \to C$;
$\vdash \lambda f\lambda g\lambda x(g)(f)x : (A \to B), (B \to C), A \to C$;

On a rapidement étendu la correspondance preuve-programmes au calcul des prédicats, qui est beaucoup plus expressif, puisqu'on peut y formaliser la théorie des ensembles, et donc l'ensemble des mathématiques. Comment fait-on ?

On prend des *variables d'individu* $x, y, \ldots$ qui représentent des ensembles, les symboles logiques $\bot, \to, \forall$ et un symbole de relation $\in$ (en général, il y en a plusieurs, mais je vais m'intéresser seulement à la théorie des ensembles). Les formules de théorie des ensembles sont construites avec ces trois règles :

- $\bot$, $x \in y$ sont des formules, appelées *formules atomiques* ;
- Si A, B sont des formules, alors $A \to B$ est une formule ;
- Si A est une formule et x une variable, alors $\forall x\, A$ est une formule.

Par exemple $\forall z(z \in x \to z \in y)$ est une formule, notée $x \subset y$.
La formule $(x \subset y, y \subset x \to \bot) \to \bot$ sera notée $x = y$. C'est ce qu'on appelle l'*égalité extensionnelle*.

Les connecteurs usuels de la logique sont considérés comme des abréviations :
$\neg A$ est $A \to \bot$; $A \wedge B$ est $(A, B \to \bot) \to \bot$; $A \vee B$ est $\neg A, \neg B \to \bot$; $\exists x\, A$ est $\neg\forall x \neg A$.

On peut ainsi écrire dans ce langage tous les axiomes de ZF. Par exemple, *l'axiome d'extensionnalité* s'écrit : $\forall x \forall y[\forall z(z \in x \leftrightarrow z \in y) \to \forall z(x \in z \to y \in z)]$.

Remarque. La formule $\forall z(x \in z \to y \in z)$ exprime l'égalité de Leibniz : les ensembles x et y ont les mêmes propriétés. Elle implique l'égalité extensionnelle, à l'aide du *schéma de compréhension* et de l'*axiome de la paire*. L'axiome d'extensionnalité exprime la réciproque. Quand on construit un modèle de ZF, c'est souvent l'axiome le plus difficile à satisfaire.

Les règles de déduction sont celles de la logique propositionnelle intuitionniste, complétées par les règles suivantes pour le quantificateur universel $\forall$:

5. $A_1, \ldots, A_n, \forall x\, A \vdash A[y/x]$ pour toute variable y.
6. $A_1, \ldots, A_n, A \vdash \forall x\, A$ si x n'est pas dans $A_1, \ldots, A_n$.

Ces règles du *calcul des prédicats intuitionniste* peuvent aussi être "décorées" par des λ-termes, avec une facilité étonnante :

5. $x_1 : A_1, \ldots, x_n : A_n, z : \forall x\, A \vdash z : A[y/x]$.
6. $x_1 : A_1, \ldots, x_n : A_n, z : A \vdash z : \forall x\, A$ si x n'est pas dans $A_1, \ldots, A_n$.

Cette syntaxe suffit pour écrire les axiomes et les théorèmes de ZF, mais elle pose rapidement des problèmes pratiques insurmontables, à cause de la longueur des formules. C'est pourquoi on lui ajoute des *symboles de fonctions* ; par exemple, $\{x, y\}$ est un symbole à deux arguments qui représente la paire x, y ; le symbole $\mathscr{P}(x)$ n'a qu'un argument et représente l'ensemble des parties de x. Pour chacun de ces symboles, on ajoute un axiome à ZF :
$\forall x \forall y \forall z (z \in \{x, y\} \leftrightarrow z = x \vee z = y)$; $\forall x \forall y (y \in \mathscr{P}(x) \leftrightarrow y \subset x)$.
Par superposition, ces symboles permettent d'écrire des *termes*, par exemple $\{\{x, x\}, \{x, y\}\}$ (qui représente le couple) ou $\mathscr{P}(\{\mathscr{P}(x), \{\mathscr{P}(\{x, y\})\}\})$. Cela abrège énormément les formules.
Il faut alors remplacer la règle 5, par la suivante :

5. $A_1, \ldots, A_n, \forall x\, A \vdash A[t/x]$ pour tout terme t.

et, après "décoration" :

5. $x_1 : A_1, \ldots, x_n : A_n, z : \forall x\, A \vdash z : A[t/x]$ pour tout terme t.

On obtient ce qu'on appelle une *extension conservative* de ZF ; cela veut dire que, si on démontre dans cette théorie une formule du langage de ZF (ne comportant pas les nouveaux symboles), alors on peut la démontrer dans ZF.

Remarque. Bien qu'on utilise le même mot, attention à ne pas confondre les termes du calcul des prédicats et ceux du λ-calcul (qu'on appelle plutôt λ-termes). Le contexte indique toujours clairement de quoi il s'agit.

Toute démonstration faite avec les règles 1 à 6 de la logique intuitionniste donne donc un programme sous la forme d'un λ-terme. Il se pose maintenant deux questions essentielles :

i) Ces règles ne permettent pas de formaliser les démonstrations usuelles, car les preuves mathématiques utilisent la logique *classique* et non pas intuitionniste ; et, en plus, elles nécessitent des *axiomes*, par exemple ceux de l'arithmétique, ou de l'analyse, ou de la théorie des ensembles (de Zermelo ou même de Zermelo-Frænkel avec axiome du choix).

ii) Comment doit-on exécuter les programmes obtenus qui sont des λ-termes ? Il faut fixer une *stratégie de réduction* pour les sous-termes de la forme $(\lambda x\, t)u$, qu'on appelle des *redex*. De plus, si le programme contient des variables libres $x_1, \ldots, x_n$, l'exécution se bloque sur des termes de la forme $(x_i)u$.

Ces deux problèmes ont été traités de façon pas très satisfaisante :

i) On passe de la logique intuitionniste à la logique classique en ajoutant un seul axiome, le *tiers exclu* qui est $(A \to B), (\neg A \to B) \to B$.
Un axiome équivalent plus simple est le *raisonnement par l'absurde* ou *loi de Peirce* qui est :

7. $A_1, \ldots, A_n \vdash (\neg A \to A) \to A$.

Si on veut formaliser des preuves, par exemple en théorie des ensembles avec axiome du choix, il faut ajouter la règle :

8. $A_1, \ldots, A_n \vdash Z$ pour chaque axiome Z de ZFC.

Pendant fort longtemps, la réponse à la question (i) a consisté à éliminer purement et simplement les axiomes supplémentaires. La méthode est différente pour le tiers exclu et pour les axiomes de ZF.

Pour éliminer les axiomes de ZF, on se contente de les faire passer à gauche du symbole $\vdash$ c'est-à-dire qu'on les met en hypothèse. C'est une méthode triviale, qui ne règle rien comme on va le voir.

Pour éliminer le tiers exclu, on utilise donc une méthode plus subtile, due à Gödel, qui consiste à simuler la logique classique dans la logique intuitionniste :
si on peut montrer $A_1, \ldots, A_k \vdash A$ en logique classique, alors on peut montrer $A'_1, \ldots, A'_k \vdash A'$ en logique intuitionniste. La formule A' est obtenue en remplaçant, dans A, chaque sous-formule *atomique* $t \in u$ par sa double négation $\neg\neg(t \in u)$. Elle est donc équivalente à A, en logique classique.

De cette façon si on a une preuve classique d'une formule A à partir d'axiomes $A_1, \ldots, A_n$ de ZFC par exemple, alors on a une preuve de $A'_1, \ldots, A'_n \vdash A'$ en logique intuitionniste. On obtient ainsi un programme pour n'importe quelle preuve mathématique et le tour est joué.

ii) Mais alors intervient la question (ii), qui va mettre en lumière les défauts de cette stratégie. Pour chaque axiome A_i de ZFC utilisé dans la démonstration et donc mis en hypothèse, on a introduit une variable libre x_i dans le programme, qui ne sera donc pas exécutable.
Pour le tiers exclu, la situation est meilleure, car on n'a pas ajouté d'hypothèse. Mais la transformation d'une preuve classique en preuve intuitionniste est pratiquement inextricable et les programmes obtenus sont, sauf rares exceptions, impossibles à interpréter.

Une solution bien meilleure serait de trouver un programme τ pour le tiers exclu, et un programme τ_Z pour chaque axiome Z de ZF, qui nous permette de "décorer" les règles 7 et 8 ci-dessus sous la forme :

7. $x_1 : A_1, \ldots, x_n : A_n \vdash \tau : (\neg A \to A) \to A$.
8. $x_1 : A_1, \ldots, x_n : A_n \vdash \tau_Z : Z$.

Bien entendu, les programmes τ et τ_Z doivent être exécutables, donc ne pas comporter de variables libres.

On a longtemps cru qu'il était impossible de trouver un programme pour le tiers exclu ; ou plutôt on ne s'est même pas posé la question. La logique intuitionniste étant traditionnellement qualifiée de "constructive", on pensait naturellement que seules pouvaient être transformées en programmes les preuves constructives. On est allé jusqu'à *démontrer* que les preuves classiques ne pouvaient pas produire des programmes.
Quant à trouver des programmes pour les axiomes de ZF, il ne fallait pas y songer : les programmes sont obtenus à partir de preuves et, comme chacun sait, on ne prouve pas les axiomes.

La correspondance de Curry-Howard en logique intuitionniste est néanmoins devenue un sujet de recherches extrêmement actif, à la fois en logique (théorie des démonstrations) et en informatique. Divers systèmes de λ-calcul typé sont apparus, dont les plus connus sont la théorie des types de Martin-Löf et le système F de Girard. On a construit sur ce principe des assistants de preuves (Coq, NuPRL, HOL, Agda,...) qui ont servi à vérifier des preuves très complexes (théorème des 4 couleurs, conjecture de Kepler) et ont même été appliqués à la preuve de programme industriels.

Mais on est encore loin de la transformation des preuves mathématiques en programmes exécutables. En effet, il manque pour cela au λ-calcul un ingrédient essentiel que, heureusement poussés par la nécessité, les informaticiens avaient déjà introduit depuis longtemps dans les langages de programmation.

En effet, si les premiers ordinateurs ont exclusivement servi à faire des calculs numériques, il y a bien longtemps que ce n'est plus du tout l'application essentielle de l'informatique. L'idée de représenter un ordinateur par une machine de Turing ou un λ-terme qui calcule une fonction récursive est tout simplement ridicule. Vous utilisez sans arrêt et sans y prêter la moindre attention, une masse gigantesque de programmes, qui sont les réseaux et les systèmes d'exploitation, dans votre ordinateur ou votre téléphone, mais aussi dans les serveurs et les routeurs qui vous répondent dans le monde entier. Ces programmes ne calculent pas de fonction récursive, ils reçoivent et ils envoient des *paquets d'information* ; ils ne font qu'interagir avec leurs interlocuteurs qui sont également des programmes, sur la même machine ou sur d'autres et aussi, bien plus rarement, des humains.
En un mot, la caractéristique essentielle des programmes est *l'interactivité*.
Le λ-calcul, ou la logique combinatoire, représentent un programme qui démarre, puis s'arrête éventuellement, mais n'interagit pas. Il leur manque donc quelque chose d'essentiel.

La correspondance preuves-programmes en logique classique

En 1990, Timothy Griffin, un jeune informaticien à Cornell University, s'aperçoit qu'une instruction sophistiquée du langage SCHEME (un dialecte de LISP) a le type de la loi de Peirce $(\neg A \to A) \to A$. Il s'agit de `call-with-current-continuation`, ou `call/cc` en bref. Il en parle à son directeur de thèse, R. Constable, qui lui rappelle que c'est impossible, car la loi de Peirce n'est pas valide en logique intuitionniste. Mais, finalement, il publie quand même un papier sur le sujet, qui fera date [1].

L'instruction `call/cc` fait une opération qui n'a pas de sens en λ-calcul : elle sauvegarde l'environnement courant et rend un pointeur appelé *continuation*. On peut alors utiliser cette continuation à tout moment pour rétablir l'environnement sauvegardé.

Tous les langages de programmation ont une instruction de ce type (par exemple `setjmp` et `longjmp` en C). Elle est indispensable pour le traitement général des exceptions, du multitâche dans un système, etc. Elle a été inventée comme substitut de `goto`, qui est inutilisable dans des programmes complexes, car ses effets sont impossibles à maîtriser.

Je me rappelle encore ma stupéfaction en apprenant le résultat de Griffin. Comment est-il possible qu'un mode de raisonnement aussi élémentaire et intuitif que le tiers exclu corresponde à une instruction aussi technique et sophistiquée que `call/cc` ? J'ai tout de suite pensé que cette découverte aurait, sur la logique, un retentissement comparable à celui du théorème de Gödel, soixante ans auparavant. Il fallait donc considérer, au lieu du λ-calcul tout seul, une structure à deux types d'objets : les *programmes* et les *environnements*.
J'ai été immédiatement convaincu que, puisque le problème du tiers exclu était résolu alors que c'était, de loin, le plus difficile, ce serait un jeu d'enfant de résoudre le problème posé par les axiomes de ZFC. C'était le cas, en effet, sauf pour deux axiomes : l'axiome du choix et, de façon plus inattendue, l'axiome d'extensionnalité.

La découverte de T. Griffin est tout à fait fondamentale, elle touche à la racine même du raisonnement mathématique. C'est pourquoi je lui attache autant d'importance qu'au théorème d'incomplétude de Gödel. Pourtant elle est restée totalement inconnue, même chez les logiciens ; sauf, bien sûr, dans le petit cercle de ceux qui s'intéressent à la correspondance de Curry-Howard. Plusieurs systèmes de λ-calcul typé sont alors apparus, qui étendent cette

correspondance à la logique classique. Le plus remarquable, et qui a eu le plus de succès, est le $\lambda\mu$-calcul de Michel Parigot.

Mais la syntaxe n'est d'aucune utilité pour savoir quels axiomes choisir pour l'arithmétique ou pour la théorie des ensembles. De la même façon, aucun système basé sur la syntaxe, c'est-à-dire sur les preuves, ne peut répondre à la question : quel programme est associé à un axiome donné, par exemple le schéma de remplacement, ou l'axiome du choix ou l'hypothèse du continu ? Il faut donc faire de la *sémantique*.
Or, il existe une sémantique pour l'arithmétique intuitionniste, qui est donnée par la *théorie de la réalisabilité* de S.C. Kleene. C'est pourquoi la théorie dont je vais parler maintenant s'appelle la *réalisabilité classique*.

Il est intéressant de noter que cette théorie, dont le but est d'étendre la correspondance preuves-programmes à la théorie des ensembles, ait donné comme sous-produit une nouvelle méthode pour construire des modèles de ZF, différente de celles de Gödel (modèles intérieurs) et Cohen (forcing). Cela permet de montrer de nouveaux théorèmes de consistance relative en théorie des ensembles. Mais nous n'en parlerons pas ici.

Cette introduction informelle est terminée, j'espère qu'elle vous a donné envie d'en savoir un peu plus ; on va donc maintenant entrer dans le vif du sujet, et donc être un peu plus technique.

La programmation en λ-calcul

On a les deux booléens $\mathbf{0} = \lambda x\lambda y\, y$ et $\mathbf{1} = \lambda x\lambda y\, x$;
le programme "si b alors x sinon y" s'écrit $(b)xy$.
L'entier 5 est $\lambda f\lambda x(f)(f)(f)(f)(f)x$ qu'on écrit $\lambda f\lambda x(f)^5 x$ ou $\overline{5}$; on définit ainsi les *entiers de Church*.
Le successeur est $s = \lambda n\lambda f\lambda x(nf)(f)x$; on peut aussi prendre $\lambda n\lambda f\lambda x(f)(n)fx$.
Cela veut dire que, par exemple $(s)\overline{5}$ se réduit à $\overline{6}$ par *β-réduction*.
L'addition, la multiplication, l'exponentielle dans les entiers sont respectivement :
add $= \lambda m\lambda n\lambda f\lambda x(mf)(nf)x$; mul $= \lambda m\lambda n\lambda f(m)(n)f$; exp $= \lambda m\lambda n\, nm$.
Le prédécesseur est beaucoup moins évident ; en voici une version, assez élégante :
pre $= \lambda n\lambda f\lambda a\big(n\lambda g\lambda b\lambda c((g)(f)b)b\big)\mathbf{0}aa$.
Un programme qui compare deux entiers m, n, en donnant le booléen $\mathbf{1}$ si $m \leq n$ et $\mathbf{0}$ sinon :
cmp $= \lambda m\lambda n((m\ \text{exp})\lambda d\,\mathbf{1})(n\ \text{exp})\lambda d\,\mathbf{0}$
Remarque. L'utilisation dans ce programme du λ-terme exp n'a rien à voir avec le calcul de l'exponentielle, car ses arguments ne sont pas des entiers. Ici, il sert seulement à permuter ses deux arguments.

La programmation dite *récursive* et donc la boucle *while* peuvent s'écrire au moyen d'un *combinateur de point fixe* Y qui est tel que $(\mathsf{Y})t$ se réduit à $(t)(\mathsf{Y})t$. On peut prendre, par exemple, le combinateur de point fixe de Turing, qui s'écrit $\mathsf{Y} = AA$ avec $A = \lambda a\lambda f(f)(a)af$.

Exemple. Programmation du logarithme entier à base 2 ;
$\log_2(n)$ est défini comme le plus petit entier i tel que $n < 2^i$. L'algorithme est le suivant :
`x=1;i=0;((while x≤n)x=2x;i=i+1;) print(i);`
Pour n fixé, on cherche donc un terme R_n tel que :
$R_n xi \succ \big((\text{cmp}\ xn)((R_n)(\text{mul})\overline{2}x)(s)i\big)\,i$; le programme pour n fixé sera $R_n\overline{1}\,\overline{0}$.
On pose donc $R_n = (\mathsf{Y})\lambda r\lambda x\lambda i\,\big((\text{cmp}\ xn)((r)(\text{mul})\overline{2}x)(s)i\big)\,i$.

Le programme final est donc :
$\log_2 = \lambda n\big((\mathsf{Y})\lambda r\lambda x\lambda i((\mathrm{cmp}\,xn)((r)(\mathrm{mul})\overline{2}x)(s)i)i\big)\overline{1}\,\overline{0}$.

On peut ainsi programmer en λ-calcul n'importe quelle fonction récursive de domaine $\mathbb{N}^k$ à valeurs dans $\mathbb{N}$.

Nous avons vu que, pour la correspondance preuves-programmes, le λ-calcul est un langage de programmation adapté à la logique intuitionniste, mais insuffisant pour la logique classique. Pour pouvoir exécuter l'instruction `call/cc`, il faut introduire des *environnements*, que nous représenterons par des *piles de λ-termes clos*.

Une machine à pile

Les termes et les piles sont définis *récursivement* comme suit :

On se donne un ensemble de *constantes de termes* ou *instructions* et un ensemble de *constantes de pile*. L'une des instructions est notée cc.
Un *λ_c-terme* est un terme *clos* du λ-calcul, pouvant contenir des constantes de terme, et des constantes supplémentaires notées k_π, où π est une pile.
Une *pile* ou *environnement* est une expression $\pi = t_0 \bullet t_1 \bullet \ldots \bullet t_n \bullet \rho$, où $t_0, \ldots, t_n$ sont des λ_c-termes et ρ une constante de pile.

Les constantes k_π sont appelées des *continuations*.
Une *quasi-preuve* est un λ_c-terme qui ne contient aucune continuation.

On ne peut exécuter un λ_c-terme t qu'en lui fournissant un environnement π. Le couple (t,π) est noté $t \star \pi$ et est appelé un *processus*. Les règles d'exécution des processus sont les suivantes :

$(t)u \star \pi \succ t \star u \bullet \pi$ (push) ;
$\lambda x\, t \star u \bullet \pi \succ t[u/x] \star \pi$ (pop) ;
$\mathsf{cc} \star t \bullet \pi \succ t \star \mathsf{k}_\pi \bullet \pi$ (save the stack) ;
$\mathsf{k}_\pi \star t \bullet \omega \succ t \star \pi$ (restore the stack).

Comment s'exécute, sur cette machine, un programme de calcul d'une fonction récursive ? Prenons comme exemple la fonction $x \mapsto x^2$. Le programme est $\lambda n\lambda f(n)(n)f$ comme on l'a vu plus haut. Mais il ne peut fonctionner que si on lui fournit un environnement adéquat. C'est ce qui se passe toujours en informatique : un programme d'application ne peut s'exécuter que dans l'environnement fourni par le *système d'exploitation*. Ici, heureusement, l'environnement est très simple, mais pas du tout évident. C'est la pile $\nu \bullet T \bullet \kappa \bullet \overline{0} \bullet \rho$ où ν est l'entier dont on veut calculer le carré. Cet entier peut être lui-même donné par un programme, ce n'est pas forcément un entier de Church (appel par nom).
T est le λ-terme $\lambda g\lambda x(g)(s)x$, ρ est une constante de pile, κ une instruction qui arrête l'exécution du processus, s un terme quelconque (d'habitude, on prend le successeur sur les entiers). On obtient le résultat quand le processus s'arrête sur $\kappa \star (s)^j\overline{0} \bullet \rho$ avec $j = i^2$.

La technique est tout à fait générale : on peut remplacer $\overline{i}$ par un terme quelconque qui lui est β-équivalent, la fonction $x \mapsto x^2$ par n'importe quelle fonction récursive et $\lambda n\lambda f(n)(n)f$ par n'importe quel programme de calcul de cette fonction.

On voit que la programmation avec cette machine pose deux problèmes au lieu d'un :

• D'abord, bien sûr, écrire le programme qui est censé satisfaire nos *spécifications* (calculer le carré d'un entier, dans l'exemple précédent).
• Mais ensuite, lui fournir un environnement convenable. Dans notre exemple, l'environnement est $T \bullet \kappa \bullet \overline{0} \bullet \rho$ qui est valable pour toutes les fonctions récursives.

Dans la vie réelle, ce ne sont pas les mêmes personnes qui résolvent ces deux problèmes : si vous utilisez JAVA, par exemple, vous avez la responsabilité d'écrire votre programme, par exemple celui qui calcule la fonction récursive qui vous intéresse.
Mais l'environnement est fourni par la machine JAVA, donc par le concepteur du langage.

Nous voyons qu'il s'agit d'un jeu à deux joueurs ; ces deux personnages réapparaîtront sans cesse dans la suite, *toujours les mêmes* sous divers déguisements : ∃(loïse) et ∀(bélard), programme et environnement, client et serveur, défenseur et opposant, software et hardware, etc, etc. Et aussi, en logique, syntaxe et sémantique.

Nous allons donc, maintenant, nous attaquer à ces deux problèmes :
On va donner une technique générale pour résoudre le premier, c'est-à-dire pour écrire des programmes ;
tout simplement (si l'on peut dire) en transformant des preuves mathématiques, au moyen de la correspondance de Curry-Howard.
Le deuxième est beaucoup plus difficile, car il ne s'agit plus de syntaxe, mais de sémantique. Mais c'est par là qu'il nous faut commencer : pourquoi, en effet, nous fatiguer à écrire des programmes si on ne peut pas les faire tourner ?

Bien sûr, dans cet exposé, je ne peux pas faire le travail complètement, c'est beaucoup trop long et technique. Mais j'espère vous en donner une idée assez précise.

La réalisabilité classique

Nous allons construire un *modèle* $\mathscr{M}$ pour nos formules ; cela consiste à choisir un *domaine*, dont les éléments $a, b, \ldots$ vont être appelés *individus* ou *ensembles*. Et aussi à donner à chaque formule atomique $a \in b$ une *valeur de vérité* que nous noterons $|a \in b|$.
Dans la sémantique usuelle, appelée la *sémantique de Tarski*, ces valeurs de vérité sont 0 et 1, le faux et le vrai. A partir de là, on donne une valeur de vérité à chaque formule close :

$|\bot| = 0$;
$|A \to B| = 1$, sauf si $|A| = 1$ et $|B| = 0$;
$|\forall x\, A| = 1$ si et seulement si $|A[a/x]| = 1$ pour tout individu a.

Quand une formule F a la valeur 1, on dit que le modèle $\mathscr{M}$ *satisfait* F et on écrit $\mathscr{M} \models F$.
Si $\mathscr{M}$ satisfait les axiomes de la théorie $\mathscr{T}$ (par exemple ZF) on a un *modèle de* $\mathscr{T}$ (par exemple, un *modèle de la théorie des ensembles*).

La sémantique de Tarski a deux avantages ; d'abord sa simplicité, ensuite le théorème de complétude, qui s'énonce ainsi :

Une formule est conséquence de la théorie $\mathscr{T}$ *ssi elle est vraie dans tous les modèles de* $\mathscr{T}$.

Mais on peut penser à utiliser d'autres ensembles que $\{0,1\}$ pour les valeurs de vérité ; la seule condition est de pouvoir définir $|A \to B|$ et $|\forall x\, A|$ par récurrence. C'est ainsi que P.J. Cohen a construit des modèles de ZF qui ne satisfont pas l'axiome du choix ou l'hypothèse du continu.

Bien entendu, on ne peut pas construire un modèle de ZF à partir de rien ; le théorème d'incomplétude de Gödel nous dit qu'il faut déjà en avoir un sous la main. On peut alors le transformer en un autre qui nous convient mieux.

Comme notre sémantique est basée sur les programmes et les piles, nous allons choisir, comme valeurs de vérité, des ensembles de programmes et de piles.

Désignons par Λ_c l'ensemble des λ_c-termes et par Π l'ensemble des piles. L'ensemble des processus est $\Lambda_c \times \Pi$. Nous choisissons arbitrairement un ensemble de processus, noté $\perp\!\!\!\perp$, qui est clos par exécution inverse, ce qui veut dire que :

$$t \star \pi \in \perp\!\!\!\perp, t' \star \pi' \succ t \star \pi \Rightarrow t' \star \pi' \in \perp\!\!\!\perp.$$

On peut imaginer que $\perp\!\!\!\perp$ est l'ensemble des états que l'on cherche à atteindre, ou au contraire à éviter.

Nous allons donner à chaque formule close A, deux valeurs de vérité *qui s'opposent*, que nous notons $|A| \subset \Lambda_c$ et $\|A\| \subset \Pi$.

Imaginez deux joueurs, appelés souvent $\exists$ (ou Héloïse) et $\forall$ (ou Abélard). La première jouera des termes dans $|A|$ en affirmant que A est vraie. Le second jouera des piles dans $\|A\|$ en prétendant que A est fausse.

En fait, $|A|$ est définie à partir de $\|A\|$ par la condition :

$$t \in |A| \Leftrightarrow (\forall \pi \in \|A\|)\ t \star \pi \in \perp\!\!\!\perp.$$

On écrit $t \Vdash A$ (lire " t réalise la formule A ") au lieu de $t \in |A|$.
On définit alors $\|A\|$ par récurrence sur A :

$\|\perp\| = \Pi$;
$\|A \to B\| = \{t \bullet \pi\ ;\ t \Vdash A, \pi \in \|B\|\}$.
$\|\forall x\, A\| = \bigcup_a \|A[a/x]\|$ (a varie sur tout le domaine du modèle $\mathscr{M}$).

L'idée intuitive derrière ces définitions est que :
Pour $\perp$, il faut donner le maximum de munitions à Abélard, c'est pourquoi $\|\perp\| = \Pi$.
Pour $A \to B$, Abélard doit montrer à Héloïse que A est vraie et B fausse ; il prend donc $t \Vdash A$ et $\pi \in \|B\|$.
Pour $\forall x\, A$, Abélard doit montrer que $A[a/x]$ est fausse pour au moins un individu a.

On voit qu'il ne reste plus qu'à définir $\|a \in b\|$ pour tous les individus a, b. On le fait par induction ordinale sur les *rangs* des ensembles a, b. C'est la partie un peu technique de la définition, et nous ne l'expliciterons pas ici.

Quel rapport y a-t-il entre la sémantique et la syntaxe ? c'est ce que dit le *lemme d'adéquation* :

Lemme 1 (Lemme d'adéquation)**.**
Si $x_1 : A_1, \ldots, x_n : A_n \vdash t : B$ *et si* $u_1 \Vdash A_1, \ldots, u_n \Vdash A_n$, *alors* $t[u_1/x_1, \ldots, u_n/x_n] \Vdash B$.

Ce lemme est valable, indépendamment du choix de l'ensemble $\perp\!\!\!\perp$. Autrement dit, un programme obtenu à partir d'une preuve de la formule B est un *réalisateur universel* de B.

A quoi tout cela peut-il nous servir, pour nos problèmes de programmation ? C'est que les théorèmes de ZF jouent le rôle de *spécifications* ; ils expriment, dans un langage codé, des propriétés de programmes.
Comme exemple très simple, prenons le théorème E (pour *Euclide*) :

$$E \equiv (\forall i \in \mathbb{N})(\exists p \in \mathbb{N})(p \text{ est premier et } > i).$$

Chaque démonstration de E va nous fournir un programme t qui, quand on lui passe un entier i, fournit un entier p premier et $> i$.
En effet, par cette démonstration, on obtient :
$x : P, x_1 : A_1, \ldots, x_n : A_n \vdash t : E$, où $A_1, \ldots, A_n$ sont des axiomes de ZF, et P est la loi de Peirce (nous avons prouvé E dans ZF *en logique classique*).
Or, notre définition de la réalisabilité nous fournit providentiellement, pour chaque axiome A de ZF, une *quasi-preuve* τ_A qui réalise A ; c'est d'ailleurs là que réside la difficulté de la définition de $\|a \in b\|$, sur laquelle nous avons passé.
Par ailleurs, le lemme 2 ci-dessous montre que cc réalise la loi de Peirce P ; c'est la découverte de Griffin. D'après le lemme d'adéquation, on a donc :

$$t[\mathsf{cc}/x, \tau_{A_1}/x_1, \ldots, \tau_{A_n}/x_n] \Vdash E$$

et le programme cherché est $t[\mathsf{cc}/x, \tau_{A_1}/x_1, \ldots, \tau_{A_n}/x_n]$.
On voit que le programme t fourni par la démonstration n'est pas directement utilisable, car il contient des variables libres $x, x_1, \ldots, x_n$ qui doivent être remplacées par des quasi-preuves convenables, fournies par la sémantique. Celles-ci font partie du *système*, c'est-à-dire qu'elles sont toujours les mêmes, indépendantes de notre programme particulier.

Voici maintenant la preuve que l'instruction `call/cc` se comporte comme nous voulons :

Lemme 2. $\mathsf{cc} \Vdash (\neg A \to A) \to A$ *pour toute formule close* A.

En effet, soient $t \Vdash (\neg A \to A)$ et $\pi \in \|A\|$. Nous devons montrer que $\mathsf{cc} \star t \cdot \pi \in \perp\!\!\!\perp$. D'après la règle d'exécution de cc, et puisque $\perp\!\!\!\perp$ est clos par exécution inverse, il suffit de montrer $t \star \mathsf{k}_\pi \cdot \pi \in \perp\!\!\!\perp$. Par hypothèse sur t, il suffit de montrer $\mathsf{k}_\pi \cdot \pi \in \|\neg A \to A\|$, autrement dit $\mathsf{k}_\pi \Vdash (A \to \bot)$. Or, si $u \Vdash A$ et $\varpi \in \Pi$, on a $\mathsf{k}_\pi \star u \cdot \varpi \succ u \star \pi \in \perp\!\!\!\perp$ par hypothèse sur u et π.
C.Q.F.D.

Si nous voulons utiliser un nouvel axiome F dans nos démonstrations, il suffit donc de trouver un programme ϕ qui le réalise, éventuellement en introduisant de nouvelles instructions. Cette technique fonctionne assez souvent, mais pas toujours. Par exemple, elle ne permet pas d'obtenir l'axiome du choix complet.
Mais elle marche parfaitement avec *l'axiome du choix dépendant*, qui est la forme de l'axiome du choix que l'on utilise le plus, et de loin ; en général, d'ailleurs, on ne s'en aperçoit même pas, croyant que l'on définit une fonction par récurrence.
Rappelons son énoncé :
Si $R \subset E^2$ *est tel que* $(\forall x \in E)\exists y\, R(x, y)$, *alors pour tout* $a \in E$ *il existe une suite* $a_n (n \in \mathbb{N})$ *telle que* $a_0 = a$ *et* $(\forall n \in \mathbb{N})\, R(a_n, a_{n+1})$.
Comme on le verra plus loin, on peut écrire un programme qui réalise cet axiome, à condition d'introduire une nouvelle instruction ς, dont la règle d'exécution est la suivante :

$$\varsigma \star t \cdot \pi \succ t \star \overline{n}_\pi \cdot \pi$$

où n_π est le numéro de la pile π dans une numérotation fixée (qu'on peut supposer récursive) de l'ensemble des piles Π.
Sans être d'emploi aussi courant que le tiers exclu, le choix dépendant est néanmoins absolument indispensable en mathématiques et tout particulièrement en analyse.
Essayez donc d'exposer les propriétés élémentaires des fonctions de variable réelle ou complexe sans l'utiliser.

Le problème de la spécification

Nous avons maintenant en main la technique pour transformer en programmes toutes les preuves de théorie des ensembles classique avec choix dépendant (ce qui constitue une immense partie des mathématiques). Nous avons aussi une machine pour les exécuter.
Malheureusement, cela ne suffit pas, loin de là !
En effet, nous savons qu'un programme ne s'exécute que dans un environnement adéquat ; il s'agit, en l'occurrence, d'une pile formée de données et de fonctions système qu'il faut fournir à notre programme pour qu'il daigne montrer ce qu'il sait faire. Or, nous n'avons pas de méthode générale pour écrire cette pile. Le seul moyen est de comprendre ce que doit faire ce programme, ce qu'on appelle sa *spécification* ; et cela, avant de pouvoir l'exécuter.
Nous avons écrit ce programme à partir d'une preuve d'un certaine formule Θ. Nous sommes confrontés à ce que j'appelle le *problème de la spécification*, qui est, sans doute, le problème le plus difficile mais aussi le plus fascinant posé par la correspondance de Curry-Howard :

Etant donné un théorème Θ *(de la théorie des ensembles avec choix dépendant), quel est le comportement commun à tous les programmes obtenus à partir des preuves de* Θ *?*

Une fois ce problème résolu, il nous sera facile de trouver l'environnement nécessaire à l'exécution de ces programmes.

Nous arrivons ici à un aspect inattendu et tout à fait étonnant de la correspondance preuves-programmes. Elle nous montre, en effet, que chaque théorème de mathématiques a *deux significations* bien différentes :
la première est immédiate, c'est ce que l'on comprend en lisant son énoncé ;
la seconde est cachée et, en général, fort difficile à trouver ; c'est la spécification du comportement des programmes associés aux preuves de ce théorème.

Il faut bien comprendre que, sauf dans des cas très simples (que nous examinons plus loin), ces deux significations *n'ont aucun rapport*. Il ne faut surtout pas chercher à déduire le sens caché du sens évident.

Un exemple frappant est fourni par la loi de Peirce $(\neg A \to A) \to A$ qui est, bien entendu, un théorème et même un axiome.
On conviendra qu'il est impossible de trouver un rapport intuitif entre le raisonnement par l'absurde (le sens évident) et la spécification de l'instruction `call/cc` (le sens caché). C'est d'ailleurs pourquoi la découverte de Griffin est si remarquable.

Où en est-on sur ce problème actuellement ? A propos des théorèmes d'arithmétique, on a quelques idées précises, mais pas du tout la solution complète.
Dès qu'on passe à l'analyse, même élémentaire, il n'y a encore pratiquement aucun résultat. Je voudrais bien que l'on me donne le sens caché du théorème de Bolzano, du théorème de Rolle, ... A mon avis, beaucoup de surprises nous attendent dans ce domaine. Cette façon de reprendre le programme de Hilbert est certainement un travail de longue haleine, mais passionnant.
Il faut rapprocher ce problème des techniques de "proof mining" (par analogie avec le "data mining") en analyse, développées en particulier par U. Kohlenbach [3].

Nous allons examiner maintenant le cas des formules arithmétiques, où l'on a des résultats intéressants.

Les formules arithmétiques

Nous avons déjà examiné un cas, avec le théorème d'Euclide sur les nombres premiers. C'est celui des formules arithmétiques de la forme $(\forall n \in \mathbb{N})(\exists p \in \mathbb{N})(f(n,p) = 1)$ où f est une fonction récursive fixée. Dans ce cas (c'est pratiquement le seul), le problème de la spécification a une solution intuitive. Si t est un programme obtenu à partir d'une preuve quelconque de cette formule, alors t calcule une certaine fonction g_t (nécessairement récursive) telle que l'on ait $(\forall n \in \mathbb{N})(f(n, g_t(n)) = 1)$. Plus précisément, on a le :

Théorème 3. *Si t est un réalisateur universel de la formule $(\forall n \in \mathbb{N})(\exists p \in \mathbb{N})(f(n,p) = 1)$, alors pour tout entier n, on a $t \star \overline{n} \cdot T \cdot \kappa \cdot \overline{0} \cdot \pi \succ \kappa \star (s)^p \overline{0} \cdot \omega$ avec $f(n,p) = 1$.*

Donc, pour trouver un entier p tel que $f(n,p) = 1$, il suffit d'exécuter le processus :
$t \star \overline{n} \cdot T \cdot \kappa \cdot \overline{0} \cdot \pi$ et d'attendre que κ arrive en tête. L'entier p cherché se trouve alors au sommet de la pile, sous la forme $(s)^p \overline{0}$.

Comment obtient-on une fonction donnée au moyen d'une preuve ? Prenons, par exemple, le logarithme à base 2.
En appliquant le théorème 3, il suffit de démontrer la formule $(\forall n \in \mathbb{N})(\exists i \in \mathbb{N})(i = g(n))$ à partir des hypothèses suivantes :
$(\forall i \in \mathbb{N})(g(2^i - 1) = i)$; $(\forall i \in \mathbb{N})(g(2^i) = i + 1)$; $(\forall m, n \in \mathbb{N})(m \leq n \to g(m) \leq g(n))$.
En effet, ces formules sont réalisées par $\lambda x\, x$.

Le problème se complique avec les formules arithmétiques de la forme :
$(\forall m \in \mathbb{N})(\exists n \in \mathbb{N})(\forall p \in \mathbb{N})(f(m,n,p) = 1)$ où f est une fonction récursive donnée.
Beaucoup de théorèmes d'arithmétique célèbres sont de cette forme, par exemple le théorème de Roth. Pour fixer les idées, je prendrai le théorème suivant, qui en est un corollaire trivial : "l'équation diophantienne $x^4 = ay^4 + b + 1$ n'a qu'un nombre fini de solutions".
On peut l'écrire comme suit : $(\forall a, b \in \mathbb{N})(\exists n \in \mathbb{N})(\forall x, y \in \mathbb{N})(x^4 = ay^4 + b + 1 \to x, y \leq n)$;
$f(a,b,n,x,y)$ est la fonction caractéristique de l'ensemble récursif

$$\{(a,b,n,x,y) \in \mathbb{N}^4 \,;\, x^4 = ay^4 + b + 1 \to x, y \leq n\}.$$

Considérons un programme t obtenu à partir d'une preuve de ce théorème. Quel peut être son comportement ? L'idée naturelle est que, pour chaque couple d'entiers (a,b), il va fournir un entier n qui est une borne pour les solutions entières de cette équation. Mais ce serait un miracle, car on ne connaît pas de telle borne. On a ici un exemple de preuve dite "non constructive", qui fait un usage essentiel du tiers exclu. Le programme t fait donc certainement aussi un usage essentiel de l'instruction `call/cc`.
La question du "contenu algorithmique" de telles preuves avait été posée bien avant l'apparition de la correspondance preuves-programmes. Elle avait été étudiée par Georg Kreisel, au moyen de sa "no counter-example interpretation" [2]. Je vais l'expliquer, en utilisant la terminologie des jeux. Les deux joueurs sont toujours la belle Héloïse et le malheureux Abélard, qui est voué à perdre, puisqu'il prétend que ce théorème est faux.

Il y a deux règles du jeu différentes, l'une adaptée à la logique intuitionniste, l'autre à la logique classique. La partie commence de la même façon dans les deux cas :
Abélard propose deux entiers a, b, qui restent les mêmes pendant toute la partie. Autrement dit, il choisit une équation (avec l'idée que l'énoncé est faux pour cette équation). Héloïse choisit alors un entier n, en disant que c'est la borne cherchée ; puis Abélard choisit deux entiers x, y en disant que c'est un contre-exemple. Héloïse prétend que non.

Dans le cas intuitionniste, c'est très simple : la partie est finie et celui qui a raison a gagné.
Dans le cas classique, qui nous intéresse ici, la règle est changée au grand avantage d'Héloïse : si elle a raison, la partie s'arrête. Sinon, elle a le droit de choisir une autre borne (plus grande, de préférence) et on recommence. Donc, Héloïse ne perd que si la partie dure indéfiniment.
Si la formule considérée est vraie (ce qui est ici le cas, mais c'est difficile à démontrer), Héloïse a une stratégie gagnante pour chacune des deux règles, mais elles sont de nature très différentes. Dans le cas de la règle intuitionniste, le seul moyen est de fournir, pour chaque couple (a,b), une borne pour les solutions de l'équation $x^4 = ay^4 + b + 1$. C'est théoriquement possible, mais pratiquement impossible.
Dans le cas de la règle classique, il y a une méthode triviale qui consiste à énumérer les entiers, sans même se préoccuper des valeurs de a,b. La partie s'arrêtera certainement mais, bien sûr, la valeur de n à cet instant n'a aucune raison d'être la borne cherchée.

Par contre, si la formule considérée est fausse, alors c'est Abélard qui a une stratégie gagnante, qui est de toujours fournir un contre-exemple, c'est-à-dire des solutions de plus en plus grandes.

Revenons maintenant au programme associé à une preuve de cette formule. Son comportement va être de fournir à Héloïse une stratégie gagnante et en fait, de jouer à sa place en gagnant à tous les coups.
Quel intérêt, me direz-vous, puisqu'il y a une stratégie gagnante triviale ? C'est que cette stratégie n'est pas vraiment utilisable, car la durée de la partie dépasse alors souvent l'âge de l'univers et, en tous cas, l'espérance de vie de nos deux joueurs. Dans la stratégie obtenue au moyen d'une preuve, la longueur d'une partie est déterminée par le nombre de fois où l'on utilise l'instruction `call/cc` : c'est à ce moment que Héloïse revient en arrière et choisit un autre entier. La longueur de la partie est donc liée au nombre de fois où l'on fait un raisonnement par l'absurde dans la preuve. En général, c'est incomparablement plus court. Par exemple, pour la formule qui nous occupe, je pense que les preuves usuelles fournissent une stratégie gagnante en quelques coups seulement.

Voyons maintenant comment on peut, concrètement, exécuter un programme θ qui réalise la formule $(\forall m \in \mathbb{N})(\exists n \in \mathbb{N})(\forall p \in \mathbb{N})(f(m,n,p) = 1)$. Nous savons que la difficulté est de lui fournir un environnement adéquat. Cet environnement va être le même pour toutes les formules de la même forme. Il ne dépend pas de la fonction f ni, bien sûr, de la preuve.
Pour le construire, nous introduisons, dans le langage de programmation, deux nouvelles instructions α, κ ; elles vont servir à remplacer Héloïse et à permettre à Abélard de jouer.
L'instruction κ arrête tout simplement la partie quand elle arrive en tête.
Quand α arrive en tête, ce doit être sous la forme $\alpha \star \overline{n} \cdot t \cdot \pi$, avec $n \in \mathbb{N}, t \in \Lambda_c$ et $\pi \in \Pi$. Cela signifie que Héloïse propose l'entier n, et le processus s'arrête pour donner à Abélard le temps de réfléchir. Il choisit alors l'entier p et l'exécution reprend son cours avec le processus $t \star \overline{p} \cdot \kappa \overline{n}\,\overline{p} \cdot \pi$.

Théorème 4. *Si $\theta \Vdash (\forall m \in \mathbb{N})(\exists n \in \mathbb{N})(\forall p \in \mathbb{N})(f(m,n,p) = 1)$ alors, pour tout $m \in \mathbb{N}$, l'exécution du processus $\theta \star \overline{m} \cdot \alpha \cdot \pi$ va s'arrêter sur $\kappa \star \overline{n} \cdot \overline{p} \cdot \pi$, avec $f(m,n,p) = 1$, quelle que soit la façon dont joue Abélard.*

Ce théorème dit exactement que le programme θ joue à la place d'Héloïse et la fait toujours gagner.

Exercice. Ecrire le programme θ qui correspond à la stratégie triviale.
Soit ϕ un programme tel que $\phi\overline{m}\,\overline{n}\,\overline{p}$ calcule le booléen $f(m,n,p)$. Alors le λ-terme suivant répond à la question : $\theta = \big((\Upsilon)\lambda t\lambda n\lambda m\lambda a(an)\lambda p\lambda h(\phi mnph)((t)(s)n)ma\big)\overline{0}$.
Ce programme ne correspond pas à une preuve de la formule considérée, mais la réalise.

On a des résultats analogues pour tous les théorèmes d'arithmétique. Toutefois, ce n'est qu'une première approche et le problème est loin d'être résolu, même pour les énoncés arithmétiques. Par exemple, il y a beaucoup de résultats intéressants et difficiles de la forme $(\forall x_1,\dots,x_n\in\mathbb{N})(f(x_1,\dots,x_n)=1)$ où f est une fonction récursive (à commencer par le théorème de Fermat) et cette méthode ne nous permet pas, pour le moment, de dire quoi que ce soit sur les spécifications correspondantes.

L'axiome du choix dépendant

On aimerait bien avoir un programme qui réalise l'axiome du choix et, en principe, on connaît une méthode pour l'écrire. Toutefois, elle implique d'énormes calculs, qu'on n'a pas envie de faire, car on n'a aucune idée du comportement du programme obtenu.
Un cas particulier intéressant, pour lequel les calculs ont été faits (ils sont déjà assez pénibles), est l'axiome : "il existe un bon ordre sur $\mathbb{R}$".
Le programme obtenu est fort compliqué, et on ne comprend pas encore précisément son comportement, bien que l'on ait quelques indications. Comme nous avons vu, le problème est de lui fournir un environnement adéquat, qu'il faut construire de toutes pièces, à partir de la formule réalisée.

On va s'intéresser ici à un autre cas particulier, dont la solution est bien plus facile. Cependant, il est très important en mathématiques, car on l'utilise souvent. C'est l'*axiome du choix dépendant*, noté habituellement DC, qui s'énonce ainsi :

$\forall x\exists y\,F(x,y)\to\exists f\big((f \text{ est une fonction de domaine } \mathbb{N})\wedge(\forall n\in\mathbb{N})F(f(n),f(n+1))\big)$.

Quelques exemples de théorèmes dépendant de DC :

- Tout ensemble infini contient un sous-ensemble dénombrable.
- La réunion d'une suite d'ensembles dénombrables est dénombrable.
- Toute fonction séquentiellement continue en un point sur un espace métrique est continue en ce point.
- Le théorème de Baire (qui lui est équivalent) ...

Imaginez, par exemple, de faire de la théorie de la mesure sans cet axiome !

Remarque. On confond souvent l'utilisation de DC avec la construction d'une fonction par récurrence sur $\mathbb{N}$. C'est pourquoi, contrairement au théorème de Zorn, son usage n'est généralement pas signalé.

Impossible de faire ici la preuve en détail, mais je vais détailler le fonctionnement des instructions qui permettent de réaliser cet axiome, car il est fort intéressant.

On introduit deux instructions notées ς et γ, qui s'exécutent comme suit :

$\varsigma\star t\bullet\pi\succ t\star\overline{n}_\pi\bullet\pi$
où n_π est le numéro de π dans une numérotation fixée de l'ensemble des piles Π.

$\gamma \star \mathsf{k}_{t\bullet\overline{n}\bullet\pi} \bullet \mathsf{k}_{t'\bullet\overline{n}'\bullet\pi'} \bullet u \bullet \varpi \succ u \star \varpi$ si $n = n'$;
$t' \star \overline{n} \bullet \pi$ si $n < n'$;
$t \star \overline{n}' \bullet \pi'$ si $n' < n$.

Ces deux instructions fonctionnent l'une avec l'autre : ς sert à introduire des *numéros de pile* que γ va comparer deux à deux.
L'instruction γ manipule des triplets $(t, \overline{n}, \pi)$ formés d'un λ-terme, d'un entier et d'une pile. Un méthode élégante pour représenter un tel triplet est d'utiliser la continuation $\mathsf{k}_{t\bullet\overline{n}\bullet\pi}$.
On considère donc deux triplets $(t, \overline{n}, \pi)$ et $(t', \overline{n}', \pi')$ et on compare n et n' ;
si $n = n'$, on ne fait rien ; sinon, on continue avec le processus $t'' \star \overline{n}'' \bullet \pi''$, où π'' est la pile qui correspond au plus petit de n et n', et t'' est le λ-terme qui correspond au plus grand.

Remarque. Ce fonctionnement fait immédiatement penser à un programme de mise à jour : par exemple, il arrive souvent que l'on écrive un papier en utilisant plusieurs ordinateurs, au bureau, chez vous, en voyage sur un portable, etc. Il est important de gérer correctement ces diverses versions de façon à utiliser toujours la bonne, en général la plus récente. On suit exactement cette instruction : π représente une version du fichier, t l'ordinateur où elle se trouve et n est sa date. On compare les dates et, si elles sont différentes, on modifie un seul fichier sur un seul ordinateur ; sinon, on ne fait rien
Un tel mécanisme de mise à jour est employé de façon intensive dans le système, en particulier dans les *caches*. Ils sont utilisés lors des échanges entre mémoires de vitesses d'accès différentes : disques, mémoire vive, processeur. On conserve le plus longtemps possible une copie des informations utiles dans la mémoire la plus rapide d'accès. Il faut alors tenir à jour ces copies.
Théoriquement, le système pourrait fonctionner sans les caches. Mais, si on les désactive, il est très fortement ralenti, au point de devenir inutilisable.
De la même façon, pour démontrer des résultats d'analyse dans $\mathbb{R}^n$ ou $\mathbb{C}^n$, on peut se passer de l'axiome DC, mais les démonstrations deviennent alors très longues et illisibles.

Comment réaliser l'axiome DC à l'aide de ces instructions ? On pense à introduire un symbole de fonction f convenable de domaine $\mathbb{N}$ et à réaliser :
$\forall x \exists y\, F[x, y] \to (\forall n \in \mathbb{N}) F[f(n), f(n+1)]$.
Cela ne marche pas, mais presque. Il faut introduire un symbole de fonction g dont la signification intuitive est $g(n) = \{f(n)\}$. On arrive alors à réaliser les formules suivantes :

i) $\lambda x \lambda y \lambda z\, x \Vdash \forall n (\forall y \in g(n+1))(\exists x \in g(n))\, F[x, y]$.
ii) $\lambda x(\varsigma)(\Upsilon) x \Vdash \forall n \left(\exists y (\exists x \in g(n))\, F[x, y] \to \exists y (y \in g(n+1))\right)$.
iii) $\gamma \Vdash \forall n \forall y \forall y' (y \in g(n+1), y' \in g(n+1) \to y = y')$.

Or, à partir de ces trois formules *et des hypothèses* $\forall x \exists y\, F[x, y]$ *et* $g(0) = \{a_0\}$, il est facile de montrer que $g(n)$ est, pour $n \in \mathbb{N}$, une suite de singletons et que si on pose $g(n) = \{a_n\}$, alors on a $F[a_n, a_{n+1}]$ pour tout entier n. Autrement dit, DC est conséquence logique, dans ZF, de ces trois formules. D'où un programme, utilisant les instructions γ et ς, qui réalise DC.

Preuve. De (ii) et $\forall x \exists y\, F[x, y]$, on déduit $\forall n (\exists x (x \in g(n)) \to \exists x (x \in g(n+1)))$.
Avec $g(0) = \{a_0\}$, on obtient $(\forall n \in \mathbb{N}) \exists x (x \in g(n))$.
Avec (iii) et $g(0) = \{a_0\}$, on en déduit $(\forall n \in \mathbb{N}) \exists! x (x \in g(n))$.
Enfin, avec (i), on obtient $(\forall n \in \mathbb{N})\, F[a_n, a_{n+1}]$.

Pourquoi donc est-on amené, pour pouvoir écrire ce programme, à introduire la suite des singletons $\{a_n\}$ et non pas la suite a_n elle-même ? C'est parce qu'on ne sait pas si la formule

$\forall x \exists y\, F[x, y]$ est vraie ou fausse, et il faut prévoir tous les cas. L'ensemble a_n peut ne plus exister si cette formule est fausse, tandis que $\{a_n\}$ existe toujours et est remplacé par $\emptyset$.

Conclusion

Pour illustrer la correspondance preuves-programmes, nous avons examiné d'un peu près le cas de deux axiomes, plus ou moins controversés dans le passé : le tiers exclu et le choix dépendant, ainsi que le fonctionnement des instructions correspondantes, qui sont couramment utilisées en programmation.
Qu'avons-nous observé ? Du côté informatique, ces instructions ne sont théoriquement pas indispensables, mais sans elles, les programmes deviendraient très compliqués à écrire correctement et très longs à exécuter. En fin de compte, elles sont réellement indispensables et ont été introduites depuis longtemps dans les langages de programmation.
Du côté logique, si on refuse d'utiliser ces axiomes, les démonstrations se rallongent considérablement et deviennent très pénibles à comprendre. En vérité, personne ne peut songer à écrire la traduction intuitionniste, par double négation, de la preuve de n'importe quel théorème non trivial.
C'est d'ailleurs pourquoi les mathématiciens ne comprennent pas, en général, que l'on ait l'idée saugrenue de contester le tiers exclu et dénient tout intérêt à la logique intuitionniste. Mais l'intuition de Brouwer que le tiers exclu était un axiome spécial était très juste et très profonde. C'est grâce à l'invention de la logique intuitionniste que la correspondance de Curry-Howard a pu apparaître, l'instruction correspondant au tiers exclu étant impossible à inventer à ce moment.
L'axiome du choix dépendant n'a, lui, jamais été vraiment critiqué. Il est, en effet, très discret : on ne remarque pas sa présence dans une démonstration et on ne lui connaît pas de conséquence paradoxale, comme pour son grand frère, l'axiome du bon ordre.
Cet axiome est moins fondamental que le tiers exclu ; cependant, si on essaie de l'éliminer, toutes les preuves d'analyse deviennent inextricables, car on ne peut plus énoncer de théorème général : par exemple, pour montrer une propriété de $\mathbb{R}^n$, on devrait faire une preuve différente, de plus en plus longue, pour chaque n.

Pour terminer, parlons brièvement d'autres aspects intéressants de ce domaine, que j'ai laissés de côté dans cet exposé et qui sont des sujets de recherche :

- L'axiome du bon ordre.

On peut écrire un programme pour l'axiome : "il existe un bon ordre sur $\mathbb{R}$". Il est très compliqué et, pour le moment, incompréhensible. Cela ne m'étonne pas, car il s'agit probablement d'un programme essentiel du noyau d'un système d'exploitation. Je travaille là-dessus avec Yves Legrangérard, et nous pensons qu'il s'agit du *scheduler* (ordonnanceur). C'est le programme qui gère le multi-tâches et choisit, à tout instant, le processus qui doit s'exécuter.
Dans cette optique, un processus est représenté par un nombre réel, le bon ordre sur $\mathbb{R}$ représente l'ordre de priorité sur les processus à un instant donné. Les contraintes à cet instant fournissent un ensemble non vide de processus et on choisit le plus petit.
A ce propos, on voit apparaître ici un principe général de l'interprétation de la logique en termes de programme :

Un concept abstrait en mathématique représente un objet concret dynamique *en informa-*

tique.

L'idée générale est qu'il est extrêmement difficile de manipuler logiquement des objets dynamiques (concrets, évidemment) et qu'on préfère, pour le raisonnement, les représenter par des concepts abstraits (évidemment statiques).
Par exemple, l'ordre de priorité sur les processus est, bien entendu, un ordre total fini ; mais il change sans arrêt. On préfère, de beaucoup, raisonner sur un objet terriblement abstrait (un bon ordre sur $\mathbb{R}$) mais qui reste tranquille !

- De nouveaux modèles de la théorie des ensembles.

Un sous-produit de cette recherche sur la relation entre logique et informatique, est qu'elle fournit un outil pour construire des modèles de ZF. C'est une extension de la méthode du forcing de Cohen, où on remplace les conditions de forcing par des programmes.
(Mal)heureusement, il s'agit d'une extension assez radicale ; un peu comme quand on dit que la théorie des groupes est une extension de celle des groupes commutatifs.
Aucune des techniques d'usage courant dans le forcing n'est plus applicable.
Par exemple, une propriété essentielle des modèles de Cohen (en fait, de tous les modèles connus de ZF) qui est la conservation des ordinaux, est perdue ici. Même les entiers sont changés ! Il y a d'ailleurs à cela une interprétation informatique, naturellement.
Ces modèles sont donc assez difficiles à manipuler. On a quand même obtenu des résultats nouveaux de consistance relative [5], sans doute hors de portée du forcing. Mais ce n'est sûrement qu'un début.

Références

[1] Timothy Griffin. *A formulæ-as-type notion of control.* Conf. record 17th A.C.M. Symp. on Principles of Progr. Lang. (1990).

[2] G. Kreisel. *On the interpretation of non-finitist proofs I.*
J. Symb. Log. 16 (1951) p. 248-26.

[3] Ulrich Kohlenbach & Paulo Oliva. *Proof mining : A systematic way of analyzing proofs in mathematics.* Proc. Steklov Inst. Math., 242 (2003) p. 136-164.

[4] Jean-Louis Krivine. *Typed lambda-calculus in classical Zermelo-Fraenkel set theory.* Arch. Math. Log., 40, 3, p. 189-205 (2001).
`http://www.pps.jussieu.fr/~krivine/articles/zf_epsi.pdf`

[5] Jean-Louis Krivine. *Realizability algebras II : new models of ZF + DC.* Log. Meth. in Comp. Sc. 8 (1 :10) (2012)