Du programme de Hilbert aux programmes informatiques

Jean-Louis Krivine

Equipe PPS, Université Paris 7, CNRS

krivine@pps.univ-paris-diderot.fr

Bordeaux, mars 2012

Le programme de Hilbert

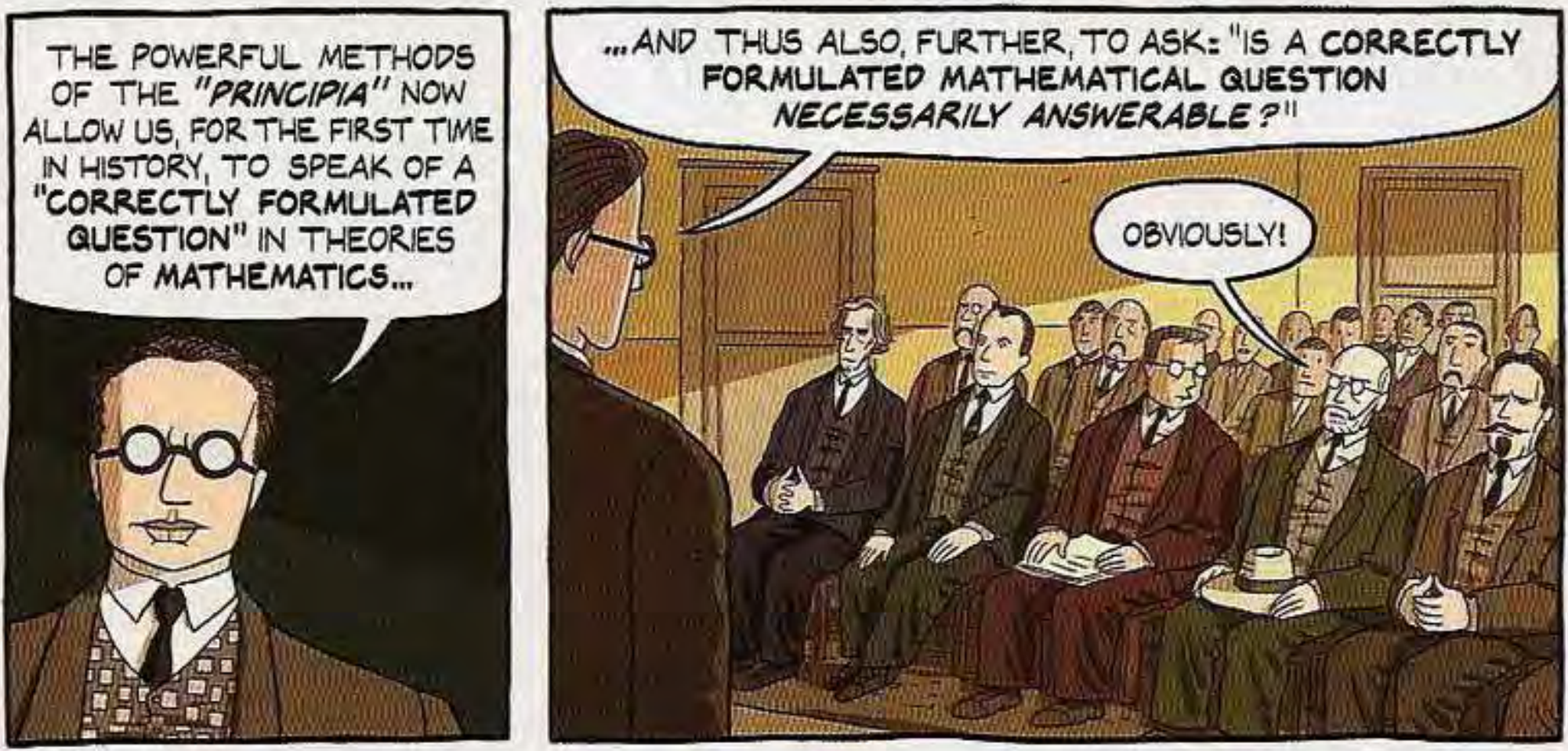
Les règles du jeu :

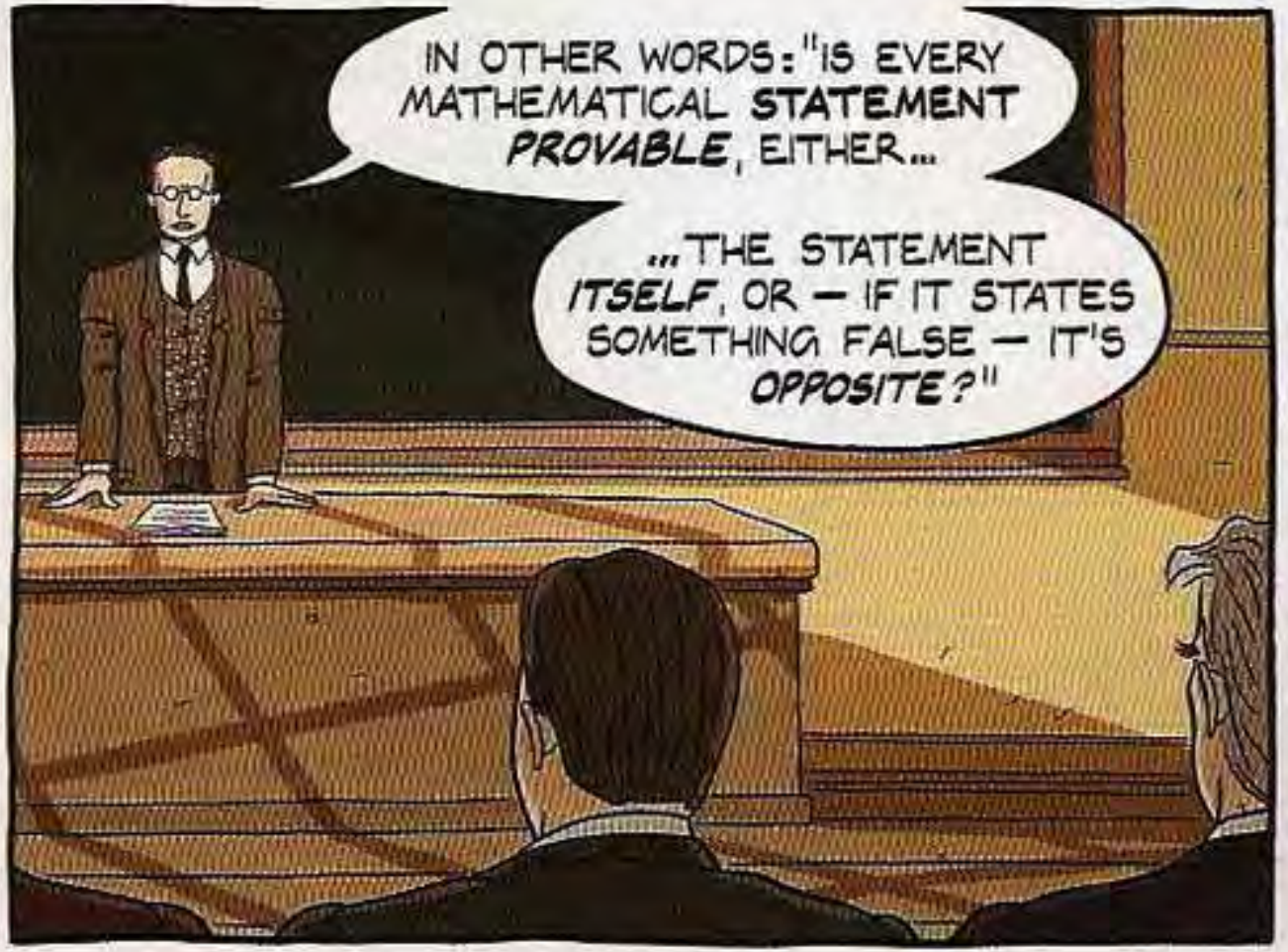
- Le théorème de complétude (Gödel 1930)
- La théorie des ensembles (Zermelo 1908, Frænkel et Skolem 1922)

On peut alors formaliser les démonstrations :
on applique les règles de la logique à partir des axiomes de ZF.

Problème : comment éviter les paradoxes (Russell, Burali-Forti, . . .) ?
Le programme de Hilbert : démontrer, par des moyens *élémentaires*,
qu'on n'obtient pas de contradiction.

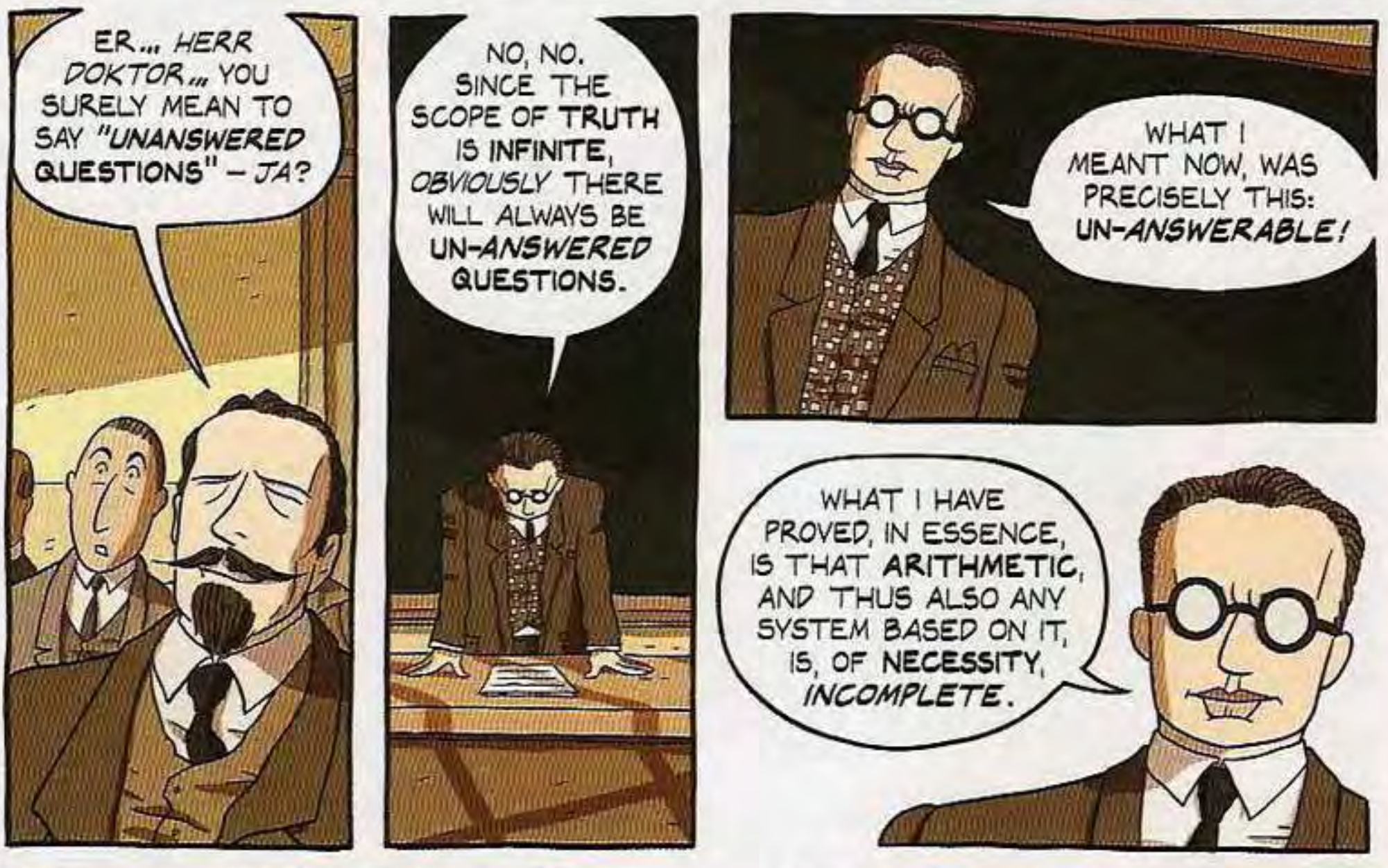
On pourra alors utiliser tranquillement les abstractions suspectes :
ensembles et bons ordres infinis, ultrafiltres, grands cardinaux, etc.
à commencer par le *tiers exclu*.
Malheureusement . . .


THE POWERFUL METHODS OF THE "PRINCIPIA" NOW ALLOW US, FOR THE FIRST TIME IN HISTORY, TO SPEAK OF A "CORRECTLY FORMULATED QUESTION" IN THEORIES OF MATHEMATICS...
...AND THUS ALSO, FURTHER, TO ASK: "IS A CORRECTLY FORMULATED MATHEMATICAL QUESTION NECESSARILY ANSWERABLE?"
OBVIOUSLY!


IN OTHER WORDS: "IS EVERY MATHEMATICAL **STATEMENT** ***PROVABLE***, EITHER...
...THE STATEMENT ***ITSELF***, OR — IF IT STATES SOMETHING FALSE — IT'S ***OPPOSITE?***"


IT IS TO THIS, MOST **FUNDAMENTAL QUESTION**, THAT I HAVE FOUND THE **ANSWER.**
WHICH IS...


NO.
OR, TO PUT IT DIFFERENTLY...
"THERE WILL ALWAYS BE UNANSWERABLE QUESTIONS!"
!
!
WHAT?
?


ER... HERR DOKTOR... YOU SURELY MEAN TO SAY "UNANSWERED QUESTIONS" – JA?
NO, NO. SINCE THE SCOPE OF TRUTH IS INFINITE, OBVIOUSLY THERE WILL ALWAYS BE UN-ANSWERED QUESTIONS.
WHAT I MEANT NOW, WAS PRECISELY THIS: UN-ANSWERABLE!
WHAT I HAVE PROVED, IN ESSENCE, IS THAT ARITHMETIC, AND THUS ALSO ANY SYSTEM BASED ON IT, IS, OF NECESSITY, INCOMPLETE.

Le théorème de Gödel

L'idée de Hilbert : les concepts abstraits (par exemple $\mathbb{N}, \aleph_1, \mathbb{R}, L^\infty, \ldots$)
ne servent qu'à aider à la manipulation d'objets concrets finis
qui se cachent derrière ces concepts.
Pour Hilbert, ces objets finis sont les *démonstrations formelles*.
Mais, sous cette forme, l'objectif est inaccessible :

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

Pourtant l'idée de Hilbert est essentiellement juste ;
mais il ne va pas assez loin (impossible à cette époque) :
il ne cherche pas à comprendre les objets mathématiques
mais seulement à sécuriser leur utilisation.
On n'y arrivera pas comme ça.

Le théorème de Gödel

Les démonstrations formelles ont des propriétés paradoxales. Exemples :

- Une théorie peut très bien démontrer sa propre contradiction

et être consistante.

Par contre, si elle démontre sa propre consistance,

elle est nécessairement contradictoire.

- Pour démontrer que la formule F est conséquence de la théorie $\mathscr{T}$

on a le droit de se servir de l'hypothèse supplémentaire

" F est conséquence de $\mathscr{T}$ ".

Etonnant, un peu inquiétant même ...

Quels sont donc ces objets concrets qui se cachent derrière $\mathbb{N}, \aleph_1, \mathbb{R}, L^\infty, \ldots$

si ce ne sont pas les démonstrations ?

Logique combinatoire et λ-calcul

Ce sont deux théories voisines, qui traitent de la *substitution des variables*,
inventées dans les années 30 par H.B. Curry et A. Church.
Elles prennent une grande importance à partir de 1958, en informatique
avec l'invention des *langages de programmation*.
Le langage LISP de J. Mc Carthy est directement issu du λ-calcul.

Or, en 1958, Curry remarque une curieuse analogie formelle entre
sa logique combinatoire et le système de règles de déduction de Hilbert.
En 1969, W. Howard fait le même rapprochement entre
le λ-calcul et la déduction naturelle de Gentzen.

Dans les deux cas, il s'agit de déduction dans un système fort peu expressif :
la logique propositionnelle intuitionniste.

Le calcul propositionnel intuitionniste

Des *variables propositionnelles* $p, q, \ldots$ dont une est notée $\bot$; un seul *connecteur* $\to$

Exemples de formules : $p \to \bot$ notée $\neg p$;

$(p \to (q \to \bot)) \to \bot$ notée $p \wedge q$;

$(p \to \bot) \to ((q \to \bot) \to \bot)$ notée $p \vee q$.

Les règles de la *déduction naturelle* :

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

On peut prouver : $\vdash A \to A$; $\vdash (A \to (B \to C)) \to ((A \to B) \to (A \to C))$;

mais pas la *loi de Peirce* : $((A \to B) \to A) \to A$ ou $(\neg A \to A) \to A$.

Le λ-calcul

On construit les *λ-termes* avec des variables $x, y, \ldots$ et deux opérations :

L'application : si t, u sont des λ-termes, $(t)u$ en est un.

L'abstraction : si x est une variable et t un λ-terme, alors $\lambda x\, t$ aussi.

On écrira souvent $(t)uv$ ou tuv pour $((t)u)v$.

Exemples : I $= \lambda x\, x$; S $= \lambda x \lambda y \lambda z (xz)(y)z$; Y $= (\lambda x \lambda y (y)(x) xy) \lambda x \lambda y (y)(x) xy$.

On écrit ainsi des *programmes* qui sont exécutés suivant la règle de 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.

La correspondance de Curry-Howard

On “décore” les preuves en déduction naturelle avec des λ-termes :

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

$t : A$ se lit *t est de type A*.

Par exemple, on obtient ainsi :

$\vdash \lambda x\, x : A \to A$; $\vdash \lambda x\, x : \bot \to A$

$\vdash \lambda x \lambda y \lambda z (xz)(y)z : (A \to (B \to C)) \to ((A \to B) \to (A \to C))$

$\vdash \lambda f \lambda g \lambda x (g)(f)x : (A \to B) \to ((B \to C) \to (A \to C))$

$\vdash \lambda x \lambda y\, yx : A \to \neg\neg A$ mais $\vdash ??? : \neg\neg A \to A$

Le calcul des prédicats

Un langage beaucoup plus expressif : celui de la théorie des ensembles.

des *variables d'individu* : $x, y, \ldots$

des *formules atomiques* : $x \in y$ et $\bot$

le *connecteur* $\rightarrow$ et le *quantificateur* $\forall x$; $\exists x$ s'écrit $\neg \forall x \neg$

Exemples de formules :

$\forall z(x \in z \rightarrow y \in z)$ notée $x = y$ (égalité de Leibniz) ;

$\forall x \forall y(\forall z(z \in x \leftrightarrow z \in y) \rightarrow x = y)$ (axiome d'extensionnalité) ;

$\forall x \forall y \exists z \forall u(u \in z \leftrightarrow (u = x \vee u = y))$ (axiome de la paire).

On complète les règles de déduction :

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 $A_1, \ldots, A_n$

Les symboles de fonction

Ils réduisent énormément la taille des formules.

Théoriquement inutiles mais pratiquement indispensables.

Exemples : $\{x, y\}$ pour la paire x, y ; $\mathscr{P}(x)$ pour l'ensemble des parties de x.

Pour chaque symbole, on ajoute un axiome à ZF :

$\forall x \forall y \forall z (z \in \{x, y\} \leftrightarrow z = x \vee z = y)$; $\forall x \forall y (y \in \mathscr{P}(x) \leftrightarrow \forall z (z \in y \rightarrow z \in x))$

Par superposition, ces symboles permettent d'écrire des *termes*

par exemple $\{\{x, x\}, \{x, y\}\}$ (qui représente le couple) ou $\mathscr{P}(\{\mathscr{P}(x), \{\mathscr{P}(\{x, y\})\}\})$.

Il faut alors remplacer la règle 5 par :

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

Cela donne une *extension conservative* de ZF.

Curry-Howard en calcul des prédicats

On “décore” les deux nouvelles règles de façon vraiment triviale :

5. $x_1 : A_1, \ldots, x_n : A_n, z : \forall x\, A \vdash z : A[t/x]$ pour tout terme t
6. $x_1 : A_1, \ldots, x_n : A_n, z : A \vdash z : \forall x\, A$ si x n’est pas dans $A_1, \ldots, A_n$

Deux (sérieux) problèmes pour obtenir des programmes utiles à partir de preuves :

- En mathématiques, on utilise la logique *classique* et non pas intuitionniste ;
  Et aussi un *système d’axiomes*, la théorie des ensembles par exemple.
- Comment doit-on exécuter les programmes obtenus ?
  Quelle stratégie de réduction faut-il employer ?
  Que faire des variables libres, s’il y en a ?

Curry-Howard en logique intuitionniste

Ces problèmes ont d'abord été réglés de façon pas très satisfaisante.
Pour les axiomes de ZF, on se contente de les laisser en hypothèse
et d'avoir des programmes avec des variables libres.
Pour la logique classique, il y a un seul axiome à ajouter :
le *tiers exclu* : $(A \to B) \to ((\neg A \to B) \to B)$
ou la *loi de Peirce* : $(\neg A \to A) \to A$

Théorème. (Gödel) Si $A_1, \ldots, A_k \vdash A$ en logique classique,
alors $A'_1, \ldots, A'_k \vdash A'$ en logique intuitionniste.
La formule A' est obtenue en remplaçant, dans A
chaque sous-formule *atomique* $t \in u$ par $\neg\neg(t \in u)$.

Théoriquement, on peut donc tout faire en logique intuitionniste.
Mais les preuves deviennent très longues et perdent tout sens intuitif.

Curry-Howard en logique classique ?

Si on veut travailler en théorie classique des ensembles
il faut ajouter deux règles de déduction :

7. $A_1, \ldots, A_n \vdash (\neg A \to A) \to A$ (loi de Peirce)
8. $A_1, \ldots, A_n \vdash Z$ pour chaque axiome Z de ZF

(où $A_1, \ldots, A_n$ sont des hypothèses inutiles).

La bonne solution serait de les "décorer" ;
trouver des *λ-termes clos* τ, τ_Z tels qu'on puisse écrire :

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

C'est impossible en restant dans le λ-calcul.
Mais la réponse existait déjà en informatique ...

Griffin et `call/cc`

En 1990, Tim Griffin, doctorant à Cornell, s'aperçoit que l'instruction
`call-with-current-continuation` (en bref `call/cc`)
du langage SCHEME (un dialecte de LISP) a le type $(\neg A \to A) \to A$.
Elle fait une opération qui n'a pas de sens en λ-calcul :
sauvegarder *l'environnement* en rendant un pointeur appelé *continuation*.
On peut ainsi rétablir cet environnement ultérieurement.

Substitut "civilisé" de l'instruction `goto` qui est incontrôlable.
Indispensable pour traiter les exceptions, le multi-tâches, etc.

Un programme n'est pas un simple calcul. Son rôle essentiel est d'*interagir*
avec son *environnement*, en échangeant des *paquets d'information*
avec d'autres programmes et (quelquefois) avec des êtres humains.

Programmation en λ-calcul

Les *booléens* : $\mathbf{0} = \lambda x \lambda y\, y$ et $\mathbf{1} = \lambda x \lambda y\, x$

le programme $\texttt{si}\ \ b\ \ \texttt{alors}\ \ x\ \ \texttt{sinon}\ \ y$ s'écrit $(b)xy$

Les *entiers de Church* : on écrit $\overline{4} = \lambda f \lambda x (f)(f)(f)(f)x$ en abrégé $\lambda f \lambda x (f)^4 x$

Le *successeur* : $\mathsf{s} = \lambda n \lambda f \lambda x (nf)(f)x$ ou bien $\mathsf{s}' = \lambda n \lambda f \lambda x (f)(n)fx$

Cela veut dire que, par exemple $(s)\overline{4}$ se réduit à $\overline{5}$ par *β-réduction*

L'*addition*, la *multiplication*, l'*exponentielle* dans les entiers :

$\mathsf{add} = \lambda m \lambda n \lambda f \lambda x (mf)(nf)x$; $\mathsf{mul} = \lambda m \lambda n \lambda f (m)(n)f$; $\mathsf{exp} = \lambda m \lambda n\, nm$

Le *prédécesseur* est beaucoup moins évident ; en voici une jolie version :

$\mathsf{pre} = \lambda n \lambda f \lambda a \big(n \lambda g \lambda b \lambda c ((g)(f)b)b\big)\mathbf{0}aa$

La *valeur booléenne de* $m \leq n$: $\mathsf{cmp} = \lambda m \lambda n \big((m\ \mathsf{exp})\lambda d\, \mathbf{1}\big)(n\ \mathsf{exp})\lambda d\, \mathbf{0}$

Noter l'utilisation du λ-terme exp qui n'a plus rien à voir avec l'exponentielle.

Programmation en λ-calcul

Le *combinateur de point fixe* de Turing : $\mathsf{Y} = AA$ avec $A = \lambda a \lambda f (f)(a) a f$

$(\mathsf{Y})\, t$ se réduit à $(t)(\mathsf{Y})\, t$

Cela donne accès à la *boucle while* et à la *programmation récursive*

Exemple. Logarithme de n : le plus petit entier i tel que $n < 2^i$

L'algorithme est : `i=0;x=1;((while x≤n)x=2x;i=i+1;) return(i);`

Pour n fixé, on cherche donc un λ-terme R_n tel que :

$\mathsf{R}_n i x \succ \big((\mathsf{cmp}\ x n)((\mathsf{R}_n)(\mathsf{mul})\overline{2} x)(\mathsf{s}) i\big)\, i$; le programme pour n fixé sera $\mathsf{R}_n \overline{0}\,\overline{1}$

On pose donc $\mathsf{R}_n = (\mathsf{Y}) \lambda r \lambda i \lambda x \big((\mathsf{cmp}\ x n)((r)(\mathsf{mul})\overline{2} x)(\mathsf{s}) i\big)\, i$

Le programme cherché est donc :

$\log_2 = \lambda n \big((\mathsf{Y}) \lambda r \lambda i \lambda x ((\mathsf{cmp}\, x n)((r)(\mathsf{mul})\overline{2} x)(\mathsf{s}) i) i\big)\, \overline{0}\,\overline{1}$

On peut ainsi programmer en λ-calcul n'importe quelle fonction récursive.

Une machine à pile

On ajoute au λ-calcul des *constantes* ou *instructions* :

une constante particulière cc ;

une infinité de constantes notées k_π, où π est une pile.

Un *λ_c-terme* est un λ-terme *clos*, pouvant contenir ces constantes.

Une *pile* ou *environnement* est une expression $\pi = t_0 \cdot t_1 \cdot \ldots \cdot t_n \cdot \rho$

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*.

Une machine à pile

Les règles d'exécution des processus :

$(t)u \star \pi \succ t \star u \bullet \pi$ (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).

Exemple de calcul d'une fonction récursive :

$n \mapsto n^2$ avec le programme $\lambda n \lambda f (n)(n) f$.

On exécute le processus : $\lambda n \lambda f (n)(n) f \star \nu \bullet \lambda g\, g \circ \mathsf{s} \bullet \kappa \bullet \overline{0} \bullet \rho$ avec $g \circ \mathsf{s} = \lambda x (g)(\mathsf{s}) x$.

κ est une *instruction d'arrêt*, ρ une pile quelconque

ν est l'entrée qui peut être un programme de type entier (appel par nom).

Le processus s'arrête sur $\kappa \star (\mathsf{s})^j \overline{0} \bullet \rho$ où j est le résultat

c'est-à-dire le carré de la *valeur* de ν.

Sémantiques de ZF

On va construire un *modèle* $\mathscr{M}$ pour nos formules ; cela consiste à :

- choisir un *domaine* M ; ses éléments $a, b, \ldots$ sont appelés *individus* ou *ensembles* ;
- donner à chaque formule atomique $a \in b$ une *valeur de vérité* notée $|a \in b|$.

Dans 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 $|B| = 0$;

$|\forall x\, A| = 1$ ssi $|A[a/x]| = 1$ pour tout individu a.

Quand une formule F a la valeur 1, on dit que $\mathscr{M}$ *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* $\mathscr{T}$.

Théorème de complétude. (Gödel)

Une formule est conséquence logique de la théorie $\mathscr{T}$ si et seulement si elle est vraie dans tous les modèles de $\mathscr{T}$.

La réalisabilité classique

C'est une autre sémantique qui utilise :

l'ensemble Λ_c des λ_c-termes ;

l'ensemble Π des piles ;

l'ensemble $\Lambda_c \star \Pi$ des processus.

Le *faux* est un ensemble $\bot\!\!\!\bot$ de processus qui est saturé, c'est-à-dire :

$$t \star \pi \in \bot\!\!\!\bot, t' \star \pi' \succ t \star \pi \Rightarrow t' \star \pi' \in \bot\!\!\!\bot$$

Une formule F a deux valeurs de vérités qui sont $|F| \subset \Lambda_c$ et $\|F\| \subset \Pi$

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

$$t \in |F| \Leftrightarrow (\forall \pi \in \|F\|)\, t \star \pi \in \bot\!\!\!\bot$$

On écrit $t \Vdash F$ (t *réalise* F) au lieu de $t \in |F|$.

Imaginer deux joueurs, ∃*loïse* qui joue un terme t et ∀*bélard* qui joue une pile π.

Elle défend la formule F, il l'attaque ; ∃loïse gagne si $t \star \pi \in \bot\!\!\!\bot$.

La réalisabilité classique

Il suffit de définir $\|F\|$ ce qu'on fait par récurrence sur F :

$\|\bot\| = \Pi$; $\|A \to B\| = \{t \cdot \pi \ ; \ t \Vdash A, \pi \in \|B\|\}$

$\|\forall x\, A\| = \bigcup_a \|A[a/x]\|$ (a décrit le domaine du modèle $\mathscr{M}$).

On peut définir $\|a \in b\|$ par induction sur les rangs de $a, b,$ de façon que :

Chaque axiome de ZF est universellement réalisé par une quasi-preuve.

Le rapport entre la sémantique et la syntaxe est donné par le

Lemme d'adéquation. Si $x_1 : A_1, \ldots, x_n : A_n \vdash t : F$ et si $u_1 \Vdash A_1, \ldots, u_n \Vdash A_n$ alors $t[u_1/x_1, \ldots, u_n/x_n] \Vdash F$. En particulier, si $\vdash t : F$ alors $t \Vdash F$.

Un terme t extrait d'une preuve de F est un *réalisateur universel* de F.

L'instruction cc est un *réalisateur universel* de la loi de Peirce :

Théorème. (Griffin) cc $\Vdash (\neg A \to A) \to A$.

L'instruction `call/cc`

Théorème. (Griffin) cc $\Vdash (\neg A \to A) \to A$.

Soient $t \Vdash (\neg A \to A)$ et $\pi \in \|A\|$. On doit montrer $\mathrm{cc} \star t \cdot \pi \in \bot\!\!\!\bot$.

D'après la règle d'exécution de cc, il suffit de montrer $t \star \mathrm{k}_\pi \cdot \pi \in \bot\!\!\!\bot$.

Par hypothèse sur t, il suffit de montrer $\mathrm{k}_\pi \cdot \pi \in \|\neg A \to A\|$ soit $\mathrm{k}_\pi \Vdash (A \to \bot)$.

Or, si $u \Vdash A$ et $\omega \in \Pi$, on a $\mathrm{k}_\pi \star u \cdot \omega \succ u \star \pi \in \bot\!\!\!\bot$ par hypothèse sur u et π.

Si on veut utiliser un nouvel axiome A dans nos démonstrations,
il suffit donc de trouver un programme ϕ qui le réalise,
éventuellement, à l'aide de nouvelles instructions.

Cela fonctionne assez souvent, mais pas toujours. Par exemple :

- les axiomes de ZF, sauf *l'extensionnalité* ;
- l'axiome du choix *dépendant*, mais pas l'axiome du choix complet.

Formules arithmétiques

Problème de la spécification. Quel est le comportement des programmes associés aux preuves d'un théorème donné ?
On s'intéresse d'abord aux théorèmes d'arithmétique.

Théorème. Si t est un réalisateur universel de la formule :
$(\forall n \in \mathbb{N})(\exists p \in \mathbb{N})(f(n,p)=1)$ où f est une fonction récursive
alors pour tout $n \in \mathbb{N}$: $t \star \overline{n} \cdot \lambda g\, g \circ \mathrm{s} \cdot \kappa \cdot \overline{0} \cdot \pi \succ \kappa \star (\mathrm{s})^p \overline{0} \cdot \omega$ avec $f(n,p)=1$.

La spécification est donc : *programme de calcul de la fonction récursive* f.

Par exemple, pour programmer la fonction $\log_2$, il suffit de
démontrer la formule $(\forall n \in \mathbb{N})(\exists i \in \mathbb{N})(i = g(n))$ avec les hypothèses :
$(\forall i \in \mathbb{N})(g(2^i - 1) = i)$; $(\forall i \in \mathbb{N})(g(2^i) = i+1)$; $(\forall m,n \in \mathbb{N})(m \leq n \to g(m) \leq g(n))$.
En effet, ces formules sont réalisées par $\lambda x\, x$.

Formules arithmétiques

Pour une formule arithmétique de la forme :

$(\forall m \in \mathbb{N})(\exists n \in \mathbb{N})(\forall p \in \mathbb{N})(f(m,n,p) = 1)$ (f récursive)

le programme réalise une *stratégie gagnante* pour ∃loïse, dans le jeu associé.

On utilise les instructions κ (arrête la partie) et α (organise le jeu).

Règle d'exécution de α : $\alpha \star \overline{n} \cdot t \cdot \pi \succ t \star \overline{p} \cdot \kappa \overline{n}\,\overline{p} \cdot \pi$

Le programme fournit l'entier n, à la place d'∃loïse ; puis ∀bélard joue l'entier p.

Théorème. 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 $m \in \mathbb{N}$, $\theta \star \overline{m} \cdot \alpha \cdot \pi \succ \kappa \star \overline{n} \cdot \overline{p} \cdot \pi$, avec $f(m,n,p) = 1$, quoi que joue Abélard.

Noter que le programme pour la stratégie triviale est :

$\theta = \big((\mathrm{Y})\lambda t \lambda n \lambda m \lambda a (an) \lambda p \lambda h (\phi m n p h)((t)(s)n)ma\big)\overline{0}$

où ϕ est tel que $\phi \overline{m}\,\overline{n}\,\overline{p}$ calcule le booléen $f(m,n,p)$.

C'est un réalisateur universel qui ne correspond à aucune démonstration.

L'axiome du choix dépendant

Un cas particulier de l'axiome du choix, sans cesse utilisé, surtout en analyse.

(DC) $\forall x \exists y\, F(x,y) \to \exists f\, (\forall n \in \mathbb{N}) F(f(n), f(n+1))$

Exemples

- Tout ensemble infini contient un sous-ensemble dénombrable.
- La réunion d'une suite d'ensembles dénombrables est dénombrable.
- La continuité séquentielle implique la continuité (espaces métriques).
- Le théorème de Baire (qui lui est équivalent) …

Il faut donc absolument réaliser cet axiome.

On se sert, pour cela, de deux nouvelles instructions notées ς et γ.

Les instructions pour DC

Leurs règles d'exécution sont les suivantes :

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

où n_π est le numéro de π dans une numérotation fixée des piles.

$$\gamma \star \mathsf{k}_t \bullet \overline{n} \bullet \pi \bullet \mathsf{k}_{t'} \bullet \overline{n}' \bullet \pi' \bullet u \bullet \omega \succ \begin{cases} u \star \omega & \text{si } n = n' \ ; \\ t' \star \overline{n} \bullet \pi & \text{si } n < n' \ ; \\ t \star \overline{n}' \bullet \pi' & \text{si } n' < n. \end{cases}$$

ς introduit des *numéros de pile* n_π ;

γ compare deux processus $t \star \pi, t' \star \pi'$ au moyen des numéros de pile $n_\pi, n_{\pi'}$

On repart avec le processus formé de la plus petite pile et du plus grand terme.

Les instructions pour DC

Ce fonctionnement est celui d'un programme de mise à jour :
π est une *version* d'un fichier, n sa *date*, t son *support* (physique ou virtuel).
Quand on compare deux versions, si les dates sont les mêmes, on ne fait rien ;
sinon, on "écrase" un des deux fichiers.

Un tel mécanisme est employé de façon intensive dans les *caches*.
Ils servent lors des échanges entre mémoires de vitesses d'accès différentes :
disques, mémoire vive, processeur.
On garde 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.
Sans les caches, le système est très fortement ralenti et inutilisable.

De même, pour démontrer des résultats d'analyse dans $\mathbb{R}^n$ ou $\mathbb{C}^n$,
on peut se passer de DC, mais les preuves sont alors très lourdes et illisibles.

L'axiome du choix dépendant

Pour réaliser DC, on pense à introduire un symbole de fonction f et à réaliser :

$\forall x \exists y\, F[x,y] \to (\forall n \in \mathbb{N}) F[f(n), f(n+1)]$.

Il faut, en fait, prendre un symbole g dont la signification est $g(n) = \{f(n)\}$.

On arrive alors à réaliser les formules suivantes :

$\lambda x \lambda y \lambda z\, x \Vdash \forall n (\forall y \in g(n+1))(\exists x \in g(n))\, F[x,y]$.

$\lambda x (\varsigma)(\Upsilon) x \Vdash \forall n \big((\exists x \in g(n)) \exists y\, F[x,y] \to \exists y (y \in g(n+1))\big)$.

$\gamma \Vdash \forall n \forall y \forall y' (y \in g(n+1), y' \in g(n+1) \to y = y')$.

Avec ces trois formules et les hypothèses $\forall x \exists y\, F[x,y]$ et $g(0) = \{a_0\}$,

on montre que $g(n)$ est une suite de singletons $\{a_n\}$ et qu'on a $F[a_n, a_{n+1}]$.

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.

Références

1. **H.B. Curry, R. Feys** *Combinatory Logic.* North-Holland, 1958.

2. **T. Griffin** *A formulæ-as-type notion of control.*
Conf. Record of the 17th A.C.M . Symp. on Principles of Progr. Languages, 1990.

3. **W. Howard** *The formulas–as–types notion of construction.*
Essays on combinatory logic, λ-calculus and formalism, Acad. Pr., 479–490, 1980.

4. **J.-L. Krivine** *Realizability in classical logic.*
Panoramas et synthèses, Société Mathématique de France, 27, 197–229, 2009.

5. **J.-L. Krivine** *Realizability algebras : a program to well order* $\mathbb{R}$.
Logical Methods in Computer Science, vol. 7, 3:02, 1–47, 2011.

6. **J.-L. Krivine** *Realizability algebras II : new models of ZF + DC.*
Logical Methods in Computer Science, vol. 8, 1:10, 2012.

Pdf files at http://www.pps.jussieu.fr/~krivine