Des maths et des programmes

Jean-Louis Krivine

Equipe PPS, Université Paris 7, CNRS

Séminaire MIM ENS Lyon

3 février 2004

La correspondance de Curry-Howard

C'est le schéma :

Démonstration $\leadsto$ Programme

Théorème $\leadsto$ Spécification

Le programme associé à une démonstration sera écrit dans un λ-calcul étendu avec de nouvelles instructions.

La spécification associée à un théorème est le comportement commun des programmes associés à ***toutes*** les preuves de ce théorème.

Cette correspondance apparaît, dans les années 60, sous une forme très limitée : *le λ-calcul typé simple intuitionniste*.

Le λ-calcul typé simple intuitionniste

Les formules sont écrites avec des variables prop. $X, Y, \ldots$ et le connecteur $\to$.

Règles de déduction :

$A_1, \ldots, A_k \vdash A_i$;

$A_1, \ldots, A_k, A \vdash B \;\Rightarrow\; A_1, \ldots, A_k \vdash A \to B$;

$A_1, \ldots, A_k \vdash A \to B$, $A_1, \ldots, A_k \vdash A \;\Rightarrow\; A_1, \ldots, A_k \vdash B$.

C'est une logique *intuitionniste* parce qu'on ne peut pas y montrer *le raisonnement par l'absurde* $(\neg A \to A) \to A$ ou $((A \to B) \to A) \to A$.

Exemple 1. $A \vdash A$; $\vdash A \to A$.

Exemple 2. $A, (A \to A) \to B \vdash A$; $(A \to A) \to B \vdash A \to A$;
$(A \to A) \to B \vdash (A \to A) \to B$; $(A \to A) \to B \vdash B$;
$\vdash ((A \to A) \to B) \to B$.

De telles démonstrations donnent des programmes écrits en λ-calcul.

Pour cela, on « décore » les règles de déduction :

Le λ-calcul typé simple intuitionniste (cont.)

$x_1{:}A_1, \ldots, x_k{:}A_k \vdash x_i{:}A_i$;

$x_1{:}A_1, \ldots, x_k{:}A_k, x{:}A \vdash t{:}B \;\Rightarrow\; x_1{:}A_1, \ldots, x_k{:}A_k \vdash \lambda x\, t{:}A \to B$;

$x_1{:}A_1, \ldots, x_k{:}A_k \vdash t{:}A \to B,\; x_1{:}A_1, \ldots, x_k{:}A_k \vdash u{:}A \;\Rightarrow$

$x_1{:}A_1, \ldots, x_k{:}A_k \vdash tu{:}B.$

Dans la suite, on écrira Γ pour noter un *contexte* $x_1{:}A_1, \ldots, x_k{:}A_k$.

Exemple 1. $x{:}A \vdash x{:}A$; $\vdash \lambda x\, x{:}A \to A$.

Exemple 2. $x{:}(A \to A) \to B, y{:}A \vdash y{:}A$;

$x{:}(A \to A) \to B \vdash \lambda y\, y{:}A \to A$;

$x{:}(A \to A) \to B \vdash x{:}(A \to A) \to B$;

$x{:}(A \to A) \to B \vdash x\lambda y\, y{:}B$;

$\vdash \lambda x\, x\lambda y\, y{:}((A \to A) \to B) \to B.$

Et les « vraies » démonstrations ?

On voudrait maintenant transformer les preuves mathématiques *usuelles* en programmes et aussi *comprendre* ce que font ces programmes.

Le système formel le plus courant pour écrire de « vraies » mathématiques s'appelle

L'analyse

Il est constitué de :

1. La logique intuitionniste du second ordre

et des axiomes suivants :

2. La loi de Peirce ou raisonnement par l'absurde $(\neg A \to A) \to A$

pour passer de la logique *intuitionniste* à la logique *classique*.

3. L'axiome de récurrence

pour parler des entiers. Le second ordre permet alors de parler des réels.

4. L'axiome du choix dépendant

On voit qu'avec *les types simples intuitionnistes*, nous sommes loin du compte.

La logique intuitionniste du second ordre

Les symboles logiques sont : $\rightarrow, \forall$,

des variables d'*individu* $x, y, \ldots$ qui représentent des entiers,

des variables de *relation* $X, Y, \ldots$ qui représentent des ensembles d'entiers,

des symboles de fonction sur les individus, p. ex. $0, s, +, \times, \ldots$

$\perp$ est défini par $\forall X\, X$; $\neg A$ par $A \rightarrow \perp$; $A \wedge B$ par $\forall X\{(A, B \rightarrow X) \rightarrow X\}$;

$\exists x\, F[x]$ par $\forall X\{\forall x(F[x] \rightarrow X) \rightarrow X\}$; $x = y$ par $\forall X(Xx \rightarrow Xy)$; etc.

Les règles de typage sont, en plus de celles des types simples :

$\Gamma \vdash t{:}A \;\Rightarrow\; \Gamma \vdash t{:}\forall x\, A$ (resp. $\forall X\, A$) si x (resp. X) n'est pas libre dans Γ.

$\Gamma \vdash t{:}\forall x\, A \Rightarrow \Gamma \vdash t{:}A[\tau/x]$ pour tout terme τ.

$\Gamma \vdash t{:}\forall X\, A \Rightarrow \Gamma \vdash t{:}A[\Phi(x_1, \ldots, x_n)/Xx_1 \ldots x_n]$ pour toute formule Φ.

Cette dernière règle est appelée *schéma de compréhension*.

Exemple. $x{:}\perp \vdash x{:}A$; $\vdash \lambda x\, x{:}\perp \rightarrow A$.

La logique classique

En 1990, Tim Griffin découvre l'interprétation de la *loi de Peirce* par l'instruction **`call/cc`** du langage SCHEME. On ajoute la règle de déduction (ou de typage)

$\Gamma,\, k{:}A \to B \vdash t{:}A \;\; \Rightarrow \;\; \Gamma \vdash \mathrm{cc}\lambda k\, t{:}A$.

Exemple 1. $x{:}A,\, k{:}\neg A \vdash kx{:}\bot$; $x{:}A \vdash \mathrm{cc}\lambda k\, kx{:}A$; $\vdash \lambda x\, \mathrm{cc}\lambda k\, kx{:}A \to A$.

Exemple 2. Preuve de $\exists x(Rx \to \forall y\, Ry)$:

$f{:}\forall x[(Rx \to \forall y\, Ry) \to X], k{:}\neg X, z{:}Rx \to \forall y\, Ry \vdash k.fz{:}\bot$

$f{:}\forall x[(Rx \to \forall y\, Ry) \to X], k{:}\neg X \vdash \mathrm{cc}\lambda zk.fz{:}Rx$ donc aussi $\forall x\, Rx$

$f{:}\forall x[(Rx \to \forall y\, Ry) \to X], k{:}\neg X \vdash \lambda d\, \mathrm{cc}\lambda zk.fz{:}Rx \to \forall y\, Ry$

$f{:}\forall x[(Rx \to \forall y\, Ry) \to X], k{:}\neg X \vdash f\lambda d\, \mathrm{cc}\lambda zk.fz{:}X$

$f{:}\forall x[(Rx \to \forall y\, Ry) \to X] \vdash \mathrm{cc}\lambda k\, f\lambda d\, \mathrm{cc}\lambda zk.fz{:}X$ et finalement

$\vdash \lambda f \mathrm{cc}\lambda k\, f\lambda d\, \mathrm{cc}\lambda zk.fz{:}\forall X(\forall x[(Rx \to \forall y\, Ry) \to X] \to X)$

L'axiome de récurrence

On définit les entiers par la formule $Ent(x) \equiv \forall X(\forall y(Xy \to Xsy), X0 \to Xx)$.
On a $\vdash \lambda f \lambda x\, f^n x{:}Ent(s^n 0)$; $\lambda f \lambda x\, f^n x$ est appelé *entier de Church* (noté $\underline{n}$).
L'axiome de récurrence est $\forall x\, Ent(x)$; il n'existe pas d'instruction pour l'interpréter.
On s'en sort en éliminant cet axiome au moyen du

Théorème. Si Φ est démontrable au moyen de l'axiome de récurrence, alors Φ^{Ent} est démontrable sans cet axiome.

Φ^{Ent} est obtenue en remplaçant, dans Φ, chaque quantificateur $\forall x \ldots$
par $\forall x(Ent(x) \to \ldots)$.

Exemple. $\Phi \equiv \forall x\, Ent(x)$; on a donc $\vdash \Phi^{Ent}$ avec
$\Phi^{Ent} \equiv \forall X \forall x(Ent(x), \forall y(Ent(y), Xy \to Xsy), X0 \to Xx)$.
On trouve, p. ex. $\vdash \lambda x \lambda f \lambda a\, x(\lambda u \lambda y \lambda z\, u(\varsigma\, y, fyz), \underline{0}, \underline{0}, a){:}\Phi^{Ent}$
avec $\vdash \varsigma = \lambda y \lambda f \lambda x\, f.yfx{:}\forall y(Ent(y) \to Ent(sy))$.

Exécuter les programmes

Comment faut-il exécuter les programmes ainsi obtenus ?

Tant que l'on reste en logique intuitionniste, il y a une réponse simple.

La β-réduction gauche ou appel par nom

repérer, dans le λ-terme, le sous-terme le plus à gauche qui est un *redex*,

c.-à-d. de la forme $(\lambda x\, t)(u)$ et le remplacer par $t[u/x]$.

Exemple. $\varsigma\, \underline{4} = (\lambda y \lambda f \lambda x\, f.yfx)\underline{4} \succ \lambda f \lambda x\, f.\underline{4}fx$.

Le redex le plus à gauche est $\underline{4}fx = (\lambda f \lambda x\, f^4 x)fx \succ f^4 x$.

Donc $\varsigma\, \underline{4} \succ \lambda f \lambda x\, f.f^4 x = \lambda f \lambda x\, f^5 x = \underline{5}$.

Mais comment faut-il exécuter l'instruction cc ? T. Griffin a pensé à

`call-with-current-continuation` du langage SCHEME,

qui met en mémoire l'environnement sous la forme d'une *continuation*.

On est conduit à étendre le lambda-calcul de façon non triviale.

Le λ_c-calcul

Λ_c (resp. Λ_c^0) est l'ensemble des *λ_c-termes* arbitraires (resp. clos).
Π est l'ensemble des *piles*. Les règles de construction sont :

1. Toute variable x, et la constante cc sont des λ_c-termes.
2. Si t, u sont des λ_c-termes et x une variable, $t(u)$ et $\lambda x\, t$ sont des λ_c-termes.
3. Si π est une pile, alors k_π est un λ_c-terme (appelé *continuation*).

Une pile est une suite $\pi = t_1. \ldots .t_n.\rho$ de λ_c-termes t_i
terminée par une *constante de pile* ρ (le *fond* de la pile) ;
$t.\pi$ dénote la pile obtenue en *empilant* t au *sommet* de π.

La constante cc est un exemple d'instruction.
On peut ajouter bien d'autres instructions, à condition de donner,
pour chacune d'elles, la *règle de réduction* correspondante.

Exécution des processus

Un *processus* est un couple : $t \star \pi$ avec $t \in \Lambda_c^0$, $\pi \in \Pi$.
On ne peut exécuter qu'un processus, pas un λ_c-terme tout seul.
t est appelé la *tête* du processus $t \star \pi$.
C'est, à chaque instant, la partie active du processus.
Règles d'exécution des processus (avec π, $\pi' \in \Pi$ et $t, u \in \Lambda_c^0$) :

$tu \star \pi \succ t \star u.\pi$	*(empiler)*	$\mathsf{cc} \star t.\pi \succ t \star k_\pi.\pi$	*(sauver la pile)*
$\lambda x\, t \star u.\pi \succ t[u/x] \star \pi$	*(dépiler)*	$k_\pi \star t.\pi' \succ t \star \pi$	*(restaurer la pile)*

Pour chaque nouvelle instruction χ, on donnera une règle de réduction.
Par exemple, si χ est une *instruction d'arrêt*, la règle est :

$\chi \star \pi \succ t \star \rho$ pour aucun processus $t \star \rho$.

Dans la suite, on utilisera une instruction χ, analogue à **`quote`** avec la règle :

$\chi \star t.\pi \succ t \star n_t.\pi$ n_t est un entier de Church

qui est le numéro du terme t dans une énumération récursive fixée de Λ_c^0.

Théorèmes et spécifications

On a vu que $I = \lambda x\, x$ et $I' = \lambda x\, \mathsf{cc}\lambda k\, kx$ ont le type $\forall X(X \to X)$.

Comment s'exécutent-ils ? Donnons-leur l'environnement (c.-à-d. la pile) $t.\pi$.

$I \star t.\pi \succ t \star \pi$; $I' \star t.\pi \succ \mathsf{cc}\lambda k\, kt \star \pi \succ k_\pi t \star \pi \succ k_\pi \star t.\pi \succ t \star \pi$.

En fait, le type $\forall X(X \to X)$ *spécifie* ce comportement :

Théorème. Si $\vdash J{:}\forall X(X \to X)$, alors $J \star t.\pi \succ t \star \pi$.

Pour chaque théorème, il se pose le *problème de la spécification*, c.-à-d. du comportement commun des λ_c-termes qui ont ce théorème pour type.

Ce problème n'est encore résolu que dans peu de cas. Deux exemples :

Théorème. Si $\vdash \theta$: « Il existe une infinité de nombres premiers », pour tout $n \in \mathbb{N}$, $\theta \star \underline{n}.\kappa.\pi \succ \kappa \star \varsigma^p \underline{0}.\pi'$ où p est un nombre premier $> n$.

Autrement dit, si on fournit au programme θ un entier n et un pointeur κ (arrêt), il va fournir, à l'adresse κ, un nombre premier $> n$.

Un théorème d'Euler. $\frac{1}{1^2}+\frac{1}{2^2}+\frac{1}{3^2}+\cdots=\frac{8}{3}(\frac{1}{1}-\frac{1}{3}+\frac{1}{5}-\cdots)^2.$

Il exprime qu'une suite de rationnels $u_n \to 0$.

Il s'écrit donc $\forall M \exists N \forall n(n > N \to |u_n| < M^{-1})$.

Le programme ne calcule pas N en fonction de M (fonction non récursive en général).

Le comportement spécifié par ce théorème est *interactif* :

le programme « défend le théorème » contre une « attaque » extérieure.

Le programme attend un entier M.

Il fournit alors un entier N_0, puis attend un entier $n_0 > N_0$.

Si $|u_{n_0}| < M^{-1}$, il s'arrête (l'attaque a échoué) ; sinon :

Il fournit un entier N_1 et ainsi de suite.

Le programme s'arrête toujours (il défait toute attaque) ; mais l'entier N obtenu peut ne pas satisfaire $\forall n(n > N \to |u_n| < M^{-1})$.

L'axiome du choix dépendant

Le théorème d'Euler est démontrable en arithmétique,
mais la preuve la plus simple est en analyse (séries de Fourier).
Elle utilise donc l'*axiome du choix dépendant ACD*.

Si $(\forall r \in \mathbb{R})(\exists r' \in \mathbb{R})F(r, r')$ alors, pour tout $r_0 \in \mathbb{R}$,
il existe une suite $r_0, r_1, \ldots r_n, \ldots$ telle que $(\forall n \in \mathbb{N})F(r_n, r_{n+1})$.

ACD est fondamental pour l'analyse :
théorème de Bolzano-Weierstrass, fonctions continues, théorie de la mesure, …

Pour l'interpréter, on ajoute une nouvelle instruction : *l'horloge*, notée $\hbar$.

Sa règle de réduction est : $\hbar \star t.\pi \succ t \star \underline{n}.\pi$

$\underline{n}$ étant l'entier de Church qui est *l'heure courante*.

Le type de $\hbar$ est un axiome ACD' plus faible que ACD en logique intuitionniste
mais équivalent en logique classique.

Le programme pour ACD utilise donc $\hbar$ et cc.

Conclusion

Les retombées en informatique

Les preuves mathématiques sont des objets syntaxiques très complexes, tout à fait comparables aux grands logiciels développés dans l'industrie. Les mathématiciens disposent d'un savoir-faire remarquable pour manipuler ces objets de façon à la fois sûre et intuitive. Le transfert de cette expertise vers l'informatique est très important, tout particulièrement dans le domaine de la fiabilité des logiciels.

Et en sciences cognitives

Depuis l'antiquité, les mathématiciens écrivent donc, en réalité, des programmes. Ou plutôt, comme l'a dit Socrate, ils les retrouvent. Où se trouvent-ils donc ? Dans le seul type d'ordinateur accessible depuis l'antiquité : le cerveau humain. La recherche des spécifications associées aux théorèmes est donc susceptible de jeter quelque lumière sur notre propre OS…