Astérisque

JEAN-YVES GIRARD

Le lambda-calcul du second ordre

Astérisque, tome 152-153 (1987), Séminaire Bourbaki,
exp. n° 678, p. 173-185

<http://www.numdam.org/item?id=SB_1986-1987__29__173_0>

© Société mathématique de France, 1987, tous droits réservés.

L'accès aux archives de la collection « Astérisque » (http://smf4.emath.fr/Publications/Asterisque/) implique l'accord avec les conditions générales d'utilisation (http://www.numdam.org/conditions). Toute utilisation commerciale ou impression systématique est constitutive d'une infraction pénale. Toute copie ou impression de ce fichier doit contenir la présente mention de copyright.

NUMDAM

Article numérisé dans le cadre du programme
Numérisation de documents anciens mathématiques
http://www.numdam.org/

Séminaire BOURBAKI
39ème année, 1986-87, n° 678

Février 1987

LE LAMBDA-CALCUL DU SECOND ORDRE

par Jean-Yves GIRARD

Une distinction classique en programmation est celle entre *langage source* et *langage objet*.

i) le *langage objet* est celui que comprend la machine ; il est lié à l'*opérationnalité*, c'est à dire à l'exécution des programmes. Il est par nature peu structuré et ne distingue pas entre bons et mauvais programmes. Le prototype théorique d'un tel langage reste le λ-calcul pur (non typé).

ii) le *langage source* permet à l'utilisateur de manipuler le langage objet ; ses instructions représentent, sous forme lisible, des complexes d'instructions du langage objet (*modules*). Pour pouvoir programmer de façon efficace, il est nécessaire de *typer* ces modules, en d'autre termes de *spécifier leurs instructions de branchement*: par exemple, si S et T sont des types de modules, $S \to T$ représentera le type des modules qui permettent de calculer des fonctions transformant un module de type S en un module de type T ; en particulier, si t est un module de type $S \to T$, si s est un module de type S, alors ts désignera le module de type T, obtenu en branchant t et s, et correspondant à l'instruction "évaluer la fonction t sur l'argument s". Les avantages d'un bon typage sont les suivants :

i) la *modularité* : les sous-unités d'un programme sont étiquetées par leur type ; leur agencement mutuel ne dépend que des types (et pas du tout du contenu de ces sous-programmes). Le typage permet de structurer les programmes ; en particulier, il en facilite l'analyse, la relecture, et permet des corrections ou des améliorations locales.

ii) la *correction* : il y a une relation très étroite entre programmes typés et démonstrations ; en fait un programme typé apparaît comme la démonstration mathématique de la correction du programme-objet sous-jacent. Par correction, on entend bien sûr la terminaison du programme, mais aussi que le résultat est bien de la forme attendue (par exemple, un programme typé "bool" doit répondre "vrai" ou "faux", pas "231").

Ces qualités deviennent cruciales pour les problèmes de programmation à grande échelle, et c'est pourquoi il y a actuellement une grande activité autour des

systèmes de λ-calcul typé, en particulier le système F (λ-*calcul du second ordre*); ces systèmes apparaissent comme des systèmes logiques intégrés (systèmes de démonstrations) dans lesquels on programme mathématiquement. Le cadre théorique général de ces systèmes est donc celui de la théorie de la démonstration : leurs propriétés essentielles viennent des symétries profondes de la logique découvertes par Herbrand et Gentzen. Bien entendu, ces langages, conçus dans le cadre d'une problématique logique théorique, ne sont pas exactement adaptés aux contraintes pratiques de la programmation, et il y encore du travail à accumuler avant de disposer d'un vrai langage de programmation ; ce travail s'organise autour de deux pôles complémentaires : l'opérationnalité, c'est à dire l'étude des étapes élémentaires du calcul, sans référence à un langage précis (amélioration du langage objet), et l'aspect spécifications, c'est à dire la mise au point de systèmes de typage complètement adaptés aux impératifs de la programmation (langage source).

I LE SYSTÈME F

Il y actuellement trois systèmes de λ-calcul typé suffisamment évolués :
i) le système F (Girard 1970, [1] , [2]), qui forme le sujet principal de cet exposé. Son intérêt principal réside dans l'utilisation de l'abstraction sur les types (opération du second ordre), qui permet d'avoir une syntaxe très simple et un très grand pouvoir expressif.
- ii) la théorie des types de Martin-Løf (1971, [3]), qui reste au premier ordre, donc avec un pouvoir expressif bien plus faible que F, mais avec des possibilités originales de typage, qui permettent d'exprimer plus de nuances que F. L'aspect syntaxique est assez lourd.
iii) la théorie des constructions de Coquand-Huet (1985, [4]), qui est -entre autres- une synthèse entre les deux systèmes susmentionnés. Il améliore le pouvoir expressif de F, tout en introduisant les nuances de typage propres au système de Martin-Løf ; dans ce système, les deux sortes d'abstraction λx. et Λα sont unifiées en une seule. La syntaxe est d'un maniement délicat.

I.1. types de F

Les types de F sont exactement les formules du calcul propositionnel du second ordre, basé sur le connecteur → et le quantificateur Λα (pour tout) :

. nous avons des variables de types : α, β, γ, ...

. si S et T sont des types nous pouvons former le type S → T

. si S est un type et α une variable de type, alors Λα.S est un type.

L'interprétation naïve des types est la suivante :

. S → T représente l'ensemble des fonctions (algorithmes) qui appliquées à un argument de type S donnent un résultat de type T.

. $\Lambda\alpha S$ représente l'ensemble des fonctions "uniformes", qui, évaluées sur un type T quelconque, prennent pour valeur un objet de type $S[T/\alpha]$.
Cette interprétation naïve ne résiste pas à un examen même superficiel ; en effet un objet de type $\Lambda\alpha.S$ doit s'appliquer à tous les types T, y compris le type $T = \Lambda\alpha.S$, et il y a un énorme problème de circularité. Reynolds, qui a redécouvert F en 1974 ([5]), a démontré récemment ([6]) que les schémas du système F sont incompatibles avec une interprétation ensembliste de $S \to T$: $S \to T$ n'est pas formé de toutes les fonctions ensemblistes de S dans T.

I.2. termes de F

Les expressions de F qui dénotent des algorithmes sont appelées *termes* ; chaque terme t a un type T bien défini, ce qu'on note $t : T$.

. pour chaque type S, on peut introduire des variables x, y, z, ... de type S ; le type peut être indiqué en indice : x^S, ou par $x : S$. Ce sont les termes les plus simples. Le reste du calcul va être basé sur une idée logique très générale : un terme t de type T, contenant des variables libres $x_1, \ldots, x_n$, de types respectifs $S_1, \ldots, S_n$, sera *exactement* une démonstration de la proposition T sous les hypothèses $S_1, \ldots, S_n$. Par exemple, $x : S$ est la démonstration triviale de S sous l'hypothèse S.

. si $t\ (= t[x])$ est un terme de type T, si x est une variable de type S, alors $\lambda x{:}S.t$ est un terme de type $S \to T$. Dans ce terme qui représente la fonction qui à x associe $t[x]$, la variable x est muette. Ce schéma correspond à la règle logique ($\to$-introduction) : d'une démonstration de T sous l'hypothèse S, passer à une démonstration de $S \to T$, sans l'hypothèse S.

. si t et u sont des termes de types respectifs $S \to T$ et S, alors tu est un terme de type T. tu représente l'application de la fonction t à l'argument u, ce qui justifie l'équation $(\lambda x{:}S.t)u = t[u/x]$ (1) . Ce schéma correspond à la règle logique ($\to$-élimination) : de $S \to T$ et S déduire T.

. si $t\ (= t[\alpha])$ est un terme de type $S\ (= S[\alpha])$, si α est une variable de types, alors $\Lambda\alpha.t$ est un terme de type $\Lambda\alpha.\mathbb{S}$. La règle est sujette à la restriction que α ne doit pas être libre dans le type d'une variable libre de t : ainsi, on peut former $\Lambda\alpha.\lambda x{:}\alpha.x$ (de type $\Lambda\alpha.\alpha \to \alpha$), qui est la fonction identique universelle, mais l'expression $\Lambda\alpha.x^\alpha$ (de type $\Lambda\alpha.\alpha$) est illégale, ce qui est naturel, vu que la variable libre x n'aurait plus de type, α étant devenu muet. $\Lambda\alpha.t$, dans lequel α est muet, représente la "fonction" qui à un type α , associe $t[\alpha]$. Ce schéma correspond à la règle logique (Λ -introduction) : de S déduire $\Lambda\alpha.\mathbb{S}$.

. si t est un terme de type $\Lambda\alpha.S$, si T est un type, alors tT est un terme de type $S[T/\alpha]$. Il s'agit de l'application de la "fonction" t au type T, ce qui nous suggère une seconde équation $(\Lambda\alpha.t)T = t[T/\alpha]$ (2). Ce schéma correspond à la

règle logique ($\wedge$-élimination) : de $\wedge\alpha.S$ déduire $S[T/\alpha]$.
Les cinq schémas écrits correspondent exactement aux règles du calcul propositionnel du second ordre, formulé dans le style de Prawitz, c'est à dire en déduction naturelle ([7]) ; bien entendu, il ne s'agit pas de la logique classique, qui ne se prête malheureusement pas à une telle approche, mais de la logique intuitionniste. L'isomorphisme général entre systèmes de déduction naturelle et systèmes fonctionnels typés est dû à Curry et surtout Howard ([8]).

I.3. pouvoir expressif de F

A priori, F ne possède pas de type pour les données informatiques courantes : booléens, listes, entiers, arbres, couples etc. ; il se trouve que le schéma permet d'écrire et de typer facilement tous ces types de données, par exemple :

. BOOLÉENS : on définit le type $\underline{bool} =_d \wedge\alpha.\alpha \to (\alpha \to \alpha)$. Les booléens $\top$ et $\bot$ sont définis par $\top =_d \wedge\alpha.\lambda x{:}\alpha.\lambda y{:}\alpha.x$, $\bot = \wedge\alpha.\lambda x{:}\alpha.\lambda y{:}\alpha.y$. Il y a de plus une possibilité de définition par cas (SI ... ALORS ... SINON) : si t et u sont des termes de type S, si v est de type $\underline{bool}$, on peut former Dtuv de type S, en posant $Dtuv =_d vStu$ (= ((vS)t)u). Calculons $Dtu\top$ au moyen des équations (1) et (2) : $(\wedge\alpha.\lambda x{:}\alpha.\lambda y{:}\alpha.x)Stu = (\lambda x{:}S.\lambda y{:}S.x)tu = (\lambda y{:}S.t)u = t$; de même $Dtu\bot = u$.

. ENTIERS : on définit le type $\underline{ent} =_d \wedge\alpha.\alpha \to ((\alpha \to \alpha) \to \alpha)$. L'entier n est représenté au moyen du terme $\bar{n} =_d \wedge\alpha.\lambda x{:}\alpha.\lambda y{:}\alpha \to \alpha.y(y(\ldots(y(x))\ldots))$ (n fois y). Le type $\underline{ent}$ permet les définitions par récurrence : si t est de type S, si u est de type $S \to S$, si v est de type $\underline{ent}$, on peut former Rtuv de type S au moyen de $Rtuv =_d vStu$. Les équations (1) et (2) nous assurent que $Rtu\bar{n} = u(u(\ldots(u(t))\ldots))$ $(= u^n(t))$.

. PRODUIT : si S et T sont des types, on définit $S \times T =_d \wedge\alpha.(S \to (T \to \alpha)) \to \alpha$; si s et t sont de types respectifs S et T, on peut former sXt de type $S \times T$ par $sXt =_d \wedge\alpha.\lambda x{:}S \to (T \to \alpha).xst$; on peut aussi former des opérations de projection de types $S \times T \to S$ et $S \times T \to T$, par $\pi^1 =_d \lambda x{:}S \times T.xS(\lambda y{:}S.\lambda z{:}T.y)$ et $\pi^2 =_d \lambda x{:}S \times T.xT(\lambda y{:}S.\lambda z{:}T.z)$, et on vérifie que les équations (1) et (2) donnent $\pi^1(sXt) = s$ et $\pi^2(sXt) = t$.

Ces exemples montrent que F, grâce à sa quantification du second ordre, sait se plier aux demandes de l'utilisateur, de manière à fournir, pour toutes les données courantes, le type et les opérateurs adéquats. Ceci dit, il faudrait avoir une réponse moins vague à la question "que peut-on au juste représenter dans F ?". Une indication utile est fournie par la caractérisation des fonctions (algorithmes) de $\mathbb{N}$ dans $\mathbb{N}$ qui sont représentables par des termes de F de type $\underline{ent} \to \underline{ent}$: on dira que le terme clos (sans variable libre) t de type $\underline{ent} \to \underline{ent}$ *représente* la fonction f de $\mathbb{N}$ dans $\mathbb{N}$ quand l'équation $t\bar{n} = \overline{f(n)}$ est démontrable pour tout n, à partir de (1) et (2) ; le théorème suivant répond à la question :

THÉORÈME 1 (Girard) : *Une fonction f de $\mathbb{N}$ dans $\mathbb{N}$ est réprésentable dans F si et seulement si il existe un algorithme (i.e. une définition récursive) pour f dont la terminaison est démontrable dans l'arithmétique de Peano du second ordre PA_2.*
La partie "seulement si" resultera du théorème 2 ; la partie "si" se démontre sans difficulté en utilisant une interprétation fonctionnelle de PA_2 dans F ([1],[2]) ou à l'aide d'une notion de réalisabilité (Martin-Løf, non publié). Si on remarque que PA_2 est un système dans lequel toutes les mathématiques courantes peuvent se formaliser, il résulte de ceci que, si nous avons une démonstration mathématique de terminaison pour un algorithme, il sera possible de trouver un terme de F de type ent → ent calculant la même fonction. Bien entendu, l'algorithme de départ et celui obtenu par normalisation à partir du terme de F (voir I.4.) peuvent différer notablement.

I.4. preuves de programmes dans F

Revenons à l'égalité de F ; les équations (1) et (2) sont avantageusement remplacées par des règles de réécriture :

$(\lambda x{:}S.t)u =/ t[u/x]$ (1') et $(\Lambda\alpha.t)T =/ t[T/\alpha]$ (2').

Ces règles de réécriture sont un mode de calcul : en itérant (1') et (2'), on espère arriver, partant d'un terme v, à une *forme normale* pour v, c'est à dire à un terme v' ne contenant plus de sous-expression sujette à réécriture. Le système de réécriture a une propriété classique de *confluence*, dite encore propriété de Church-Rosser : si $t =/ t_1$ et $t =/ t_2$, alors on peut trouver t_3 tel que $t_1 =/ t_3$ et $t_2 =/ t_3$. La confluence a deux conséquences immédiates :
i) l'unicité de la forme normale d'un terme (pourvu qu'elle existe)
ii) t = u (modulo les équations (1) et (2)) ssi on peut trouver v tel que $t =/ v$ et $u =/ v$. Le théorème 2 permettra de dire que t = u ssi t et u ont même forme normale.

La réécriture (appelée *réduction*) est l'algorithme universel de F, c'est à dire qu'elle permet d'effectuer les calculs dans F. Par exemple, soit t un terme clos de type ent → ent ; on peut le transformer en un algorithme défini sur $\mathbb{N}$ et à valeurs dans $\mathbb{N}$, ainsi : donnons nous un argument $n \in \mathbb{N}$

. on écrit le terme $t\bar{n}$ de type ent

. on le réduit jusqu'à forme normale. Il est facile de voir que les seuls termes normaux et clos de type ent sont de la forme $\bar{m}$, et on a donc $t\bar{n} =/ \bar{m}$.

. le résultat est l'entier m tel que $t\bar{n} =/ \bar{m}$.

Bien entendu, cette idée opérationnelle dépend de la convergence du processus de réduction ; celle-ci est assurée par le théorème suivant :
THÉORÈME 2 (Girard) : *Dans F, tout terme a une (unique) forme normale.*
On a en fait un peu plus : tous les termes sont *fortement normalisables*, c'est à

dire que quelque soit la façon dont on apllique les réductions (1') et (2'), on arrive toujours à la forme normale.

Quelques indications quant à la démonstration sont utiles :
i) le théorème 1 (et un argument bateau de diagonalisation) montre que la terminaison de l'algorithme de normalisation ne peut pas s'établir dans PA_2, donc il est nécessaire de sortir des mathématiques courantes : en particulier, il serait vain de chercher une idée combinatoire pour démontrer le théorème 2. La seule possibilité est de poser le problème de façon suffisamment abstraite, en faisant apparaître une définition qui utilise le troisième ordre : ce sera la réductibilité.
ii) pour des systèmes où ne se posent pas les questions de circularité inhérentes au second ordre (par exemple le système de Martin-Løf susmentionné), on dispose d'un argument non-combinatoire très efficace, dû à Tait ([9]) : la calculabilité héréditaire, ou encore *réductibilité*. Elle est définie par récurrence sur le type :

. un terme t de type atomique est réductible quand il est f.n. (fortement normalisable).

. un terme t de type $S \to T$ est réductible quand, pour tout u réductible de type S, tu (qui est de type T) est réductible.

Il est facile de voir que l'ensemble Red(T) des termes réductibles de type T vérifie les conditions suivantes :
(CR 1) si $t \in Red(T)$, alors t est f.n.
(CR 2) si $t \in Red(T)$ et $t \neq u$, alors $u \in Red(T)$
(CR 3) si t ne commence ni par un λx ni par un $\Lambda\alpha$, et si tous les termes u obtenus à partir de t par une étape de réduction sont dans Red(T), alors $t \in Red(T)$.

Ceci permet de démontrer de façon simple l'analogue du théorème 2 pour les systèmes typés au premier ordre : on montre d'abord que tous les termes sont réductibles, puis on applique (CR 1). La méthode bute au second ordre ; en effet, essayons de dire qu'un terme t de type $\Lambda\alpha.S$ est réductible quand, pour tout type T, le terme tT de type $S[T/\alpha]$ est réductible; par exemple, dans le cas de $\Lambda\alpha.\alpha$, t serait réductible quand tT (de type T) est réductible pour tout T, et nous aurions besoin de connaître la réductibilité de tous les types avant de définir celle de type $\Lambda\alpha.\alpha$. C'est la raison de l'introduction des *candidats de réductibilité* ([1], [2]) : un candidat de réductibilité de type T est une définition arbitraire de la réductibilité pour ce type, i.e. un ensemble de termes de type T vérifiant (CR 1)-(CR 3). Parmi toutes ces définitions arbitraires, il y a bien sûr la vraie réductibilité de type T, que l'on cherche à définir. Le cas du $\Lambda\alpha.$ est traité ainsi :

. un terme t de type $\Lambda\alpha.S$ est réductible quand, pour tout type T et tout candidat X de type T, tT est réductible de type $S[T/\alpha]$, *la réductibilité de ce type étant définie en prenant pour définition de la réductibilité pour* T : X .

Par exemple, le terme $t = \Lambda\alpha.\lambda x{:}\alpha.x$ de type $\Lambda\alpha.\alpha \to \alpha$, est réductible car

pour tout type T, pour tout candidat X de type T, pour tout u ε X, le terme tTu est dans X, comme on le voit facilement par (CR 3). La démonstration utilise des quantifications du second ordre qui, globalement, sortent du cadre de PA_2 ; pour un terme fixé, le théorème 2 est néanmoins démontrable dans PA_2, et celà suffit à la partie "seulement si" du théorème 1. On ne connait pas d'autre démonstration du théorème 2, et les extensions du résultat à des systèmes plus généraux (Coquand-Huet, [4]) adaptent la même idée.

F se retrouve donc être un système dont tous les programmes sont prouvés une fois pour toutes par le théorème 2.

I.5. le λ-calcul sous-jacent

Le λ-calcul pur (non typé) a été introduit et étudié dans les années 30 par Church, Kleene, Rosser etc. ; il se présente comme un analogue fonctionnel de la théorie des ensembles naïve. Il n'y a pas de types et on peut former des termes à partir des variables (non typées), soit en formant λx.t, où t est un terme, soit en formant tu, à partir de deux termes t et u. L'égalité est définie par l'équation $(\lambda x.t)u = t[u/x]$ (3), qui peut se voir comme une règle de réécriture $(\lambda x.t)u \neq t[u/x]$ (3'), vu que le λ-calcul vérifie la propriété de confluence (c'est d'ailleurs pour ce calcul qu'à été établie la première propriété de ce type, le théorème de Church-Rosser). L'analogie avec la théorie des ensembles naïve est si forte qu'on peut démarquer le paradoxe de Russell :

$\{x;\sim(x \in x)\} \in \{x;\sim(x \in x)\} \leftrightarrow \sim(\{x;\sim(x \in x)\} \in \{x;\sim(x \in x)\})$

$(\lambda x.M(xx))(\lambda x.M(xx)) = M((\lambda x.M(xx))(\lambda x.M(xx)))$

(a ε b est devenu ba, {x;P} est devenu λx.P et ∼a est devenu Ma), ce qui montre que dans le λ-calcul, tout terme M a un point fixe t : t = Mt. L'existence de points fixes a deux conséquences intimement liées, l'une positive, l'autre négative

i) toutes les fonctions (algorithmes) récursives de ℕ dans ℕ, totales ou partielles, sont représentables dans le λ-calcul.

ii) en général le processus de réduction ne converge pas; en fait il est même impossible, sur l'allure d'un terme, de décider s'il a une forme normale.

On peut considérer que, d'une certaine manière, le λ-calcul est sous-jacent à F : à tout terme t de F, on peut associer un terme t^- du λ-calcul, en oubliant toutes les opérations liées aux types (le type des variables, Λα et (.)T) ; par exemple, $(\bar{n})^-$ se retrouve être $\lambda x.\lambda y.y(y(...y(x)...))$, c'est à dire la représentation standard de n dans le λ-calcul (*entiers de Church*); de plus, la parenté entre (1') et (3') font que, si t ≠ u (dans F), alors $t^- \neq u^-$, autrement dit, la traduction $(.)^-$ est un homomorphisme pour la réduction. Celà suggère l'interprétation suivante des rôles respectifs des deux systèmes :

. l'opérationnalité (ce qui se passe réellement dans la machine) est concen-

trée au niveau du λ-calcul : la machine travaille uniquement sur les termes t^-.

. l'utilisateur lui donne par contre des termes de F; le fait que t^- vienne de t de F, assure que le programme se terminera, que les spécifications seront respectées etc.. Quand la machine reçoit t, elle le compile, c'est à dire qu'elle vérifie que t a été écrit correctement dans F et mémorise t^-.

La relation entre F et le λ-calcul reproduit donc, à un niveau théorique, la relation entre langage-source et langage-objet.

Une des questions essentielles qui se posent est alors la *typabilité* : étant donné un terme u du λ-calcul, peut-on le typer, c'est à dire trouver un terme t de F tel que $u = t^-$? Une condition nécessaire est bien sûr que u soit fortement normalisable ; cette condition n'est pas suffisante, comme un exemple récent de Ronchi (manuscrit, 1986) le montre. En fait, plus intéressante est la typabilité avec un type imposé : par exemple, Maurey a introduit un λ-terme t : $t =_d \lambda x.\lambda y.((x\lambda z.0)F)((y\lambda z'.1)F)$, avec 0,1 entiers de Church, et $F =_d \lambda f.\lambda g.gf$. Appliqué à deux entiers de Church n et m, tnm se réduit soit en 0 (si $n \leq m$), soit en 1 (si $n > m$), mais surtout, la structure symétrique du terme permet de faire le calcul en un nombre d'étapes inf(n,m). La question naturelle de typabilité est de savoir si ce terme peut se typer dans F, avec le type $\underline{ent} \to (\underline{ent} \to \underline{ent})$; Krivine a récemment montré que la réponse est négative [10].

I.6. sémantique dénotationnelle de F

L'étude fine du typage, de la normalisation etc., mettent en avant des imperfections petites ou grandes de la syntaxe, en grande partie dues à la nécessité d'écrire les instructions sous forme sequentielle; par exemple, deux termes qui ne diffèrent que par l'ordre d'application de certaines règles seront peut être moralement égaux, mais la syntaxe, en nous obligeant à choisir un ordre pour l'utilisation des règles, les différencie irréversiblement. Pour découvrir les schémas, les types, les équations, etc. "oubliés" par la syntaxe, on a recours à des méthodes sémantiques. On peut envisager deux types de sémantiques :

i) des sémantiques *opérationnelles* : de telles sémantiques devraient expliquer, sans référence à un langage précis, ce qui se passe pendant l'exécution d'un programme, par exemple en donnant une interprétation géométrique du processus de normalisation. On n'a pour l'instant que des indications partielles sur ce que pourrait être une sémantique opérationnelle de F ; une réponse complète pourrait avoir des conséquences jusque dans l'architecture des machines.

ii) des sémantiques *dénotationnelles* : beaucoup plus faciles à obtenir, elles se contentent de modéliser le résultat des calculs (et non son éxécution) : si $t =/ u$ alors t et u doivent avoir la même interprétation.

L'idée essentielle des sémantiques dénotationnelles est celle d'information

finie, qui exprime en quelque sorte la "continuité" des algorithmes fonctionnels. C'est ainsi que Scott a obtenu en 1969 le premier modèle non syntaxique du λ-calcul (voir [11] pour la version "systèmes d'information" de la sémantique de Scott). La sémantique de Scott a été simplifiée par le concept de fonction *stable* (Berry, [12]) et d'*espace cohérent* (Girard [13]), ce qui permet de donner une sémantique dénotationnelle simple pour F.

Un *espace cohérent* X est la donnée d'un ensemble $|X|$ et d'une relation binaire notée $x \frown y$ [mod X] sur $|X|$, réflexive et symétrique. Un objet a de X (notation $a \in X$) est un sous-ensemble de $|X|$ tel que : $\forall x \forall y\ (x,y \in a \rightarrow x \frown y \text{ [mod X]})$. L'interprétation de F est basée sur les principes : - les types sont des espaces cohérents - les termes d'un type donné sont les objets de l'espace cohérent associé.

Si X et Y sont des espaces cohérents, une fonction qui à tout objet de X associe un objet de Y sera dite *stable* quand elle vérifie :

(ST 1) $a \cup b \in X \rightarrow f(a \cap b) = f(a) \cap f(b)$

(ST 2) f commute aux unions filtrantes.

Si f est une fonction stable, on peut lui associer l'ensemble Tr(f), formé des couples (a,z), avec a objet fini de X, $z \in f(a)$ et tels que si $b \subset a$ et $z \in f(b)$, alors b = a. Il est facile de voir que l'application $f \mapsto Tr(f)$ est une bijection entre l'ensemble des fonctions stables de X vers Y et l'ensemble des objets d'un espace cohérent noté $X \rightarrow Y$. Le passage de f à Tr(f) permet d'interpréter le $\lambda x.$ et le passage inverse l'application.

On peut considérer les espaces cohérents comme une catégorie COH, en prenant pour morphismes les plongements. Dans ces conditions, un type dépendant d'une variable libre α va apparaître comme un foncteur de COH dans COH, par exemple $T(X) = X \rightarrow X$ etc. ; ces foncteurs préservent les limites directes et les produits fibrés. Si T est le foncteur associé à un tel type variable, un *objet de type variable* T sera une famille (a_X) indexée par les espaces cohérents, telle que

(OV 1) $a_X \in T(X)$ pour tout X

(OV 2) si f est un morphisme de X vers Y, alors $a_X = T(f)^{-1}(a_Y)$

On définit Tr(T) comme l'*ensemble* des couples (X,z) formés d'un espace cohérent fini X et de $z \in |T(X)|$, et tels que si z peut s'écrire T(f)(z') pour un morphisme f de but X, alors f est bijective, ces couples étant choisis à isomorphisme près. Si a est un objet de type variable T, on peut lui associer l'ensemble $Tr(a) = \{(X,z) \in Tr(T); z \in a_X\}$. Il est facile de voir que le passage de a à Tr(a) est une bijection entre l'ensemble des objets de type variable T et l'ensemble des objets d'un espace cohérent, noté $\Lambda\alpha.T(\alpha)$. Le passage de a à Tr(a) interprète le schéma $\Lambda\alpha.$, et le passage inverse le schéma (.)T. Les propriétés fonctorielles (limites directes, produits fibrés) qui permettent d'étendre une fonction définie au départ sur les seuls types finis, répondent à l'idée vague d'"uniformité" (I.1).

II DÉVELOPPEMENTS RÉCENTS

Les travaux en cours (notamment au sein de l'équipe de Logique de Paris VII) aboutissent à une remise en chantier des bases-mêmes du λ-calcul typé.

II.1. la logique linéaire

Le λ-calcul, typé ou non, s'accommode mal du parallèlisme; en effet, l'approche fonctionnelle impose une asymétrie entre entrées et sorties (par exemple, plusieurs arguments, mais une seule valeur), qui empêche de penser la communication dans un cadre symétrique, réversible. Cette asymétrie est inhérente à la logique intuitionniste, et on constate qu'un cadre logique classique serait plus satisfaisant. Malheureusement les propriétés structurelles des démonstrations classiques ne se prètent pas à une approche algorithmique. Ceci dit, dans le λ-calcul, dans le système F, existent des linéaments de parallèlisme : rappelons la structure symétrique du terme de Maurey (voir I.5.). De même, dans F, si f est de type $S \to T$, on peut construire $g = \lambda x{:}T \to B.\lambda y{:}S.x(fy)$ de type $(T \to U) \to (S \to U)$, qui "transpose" f : en quelque sorte g inverse dans f entrées et sorties. Le cadre logique intuitionniste empêche de formuler cette transposition sous forme satisfaisante, c'est à dire sous forme involutive. Or il se trouve que la sémantique cohérente (I.6.) suggère une décomposition des opérations logiques en opérations plus primitives : par exemple le type $S \to T$ se décompose en $!S \multimap T$, où

. $!S$ correspond à une mise en mémoire, permettant d'utiliser plusieurs fois un même argument de type S (analogue d'une algèbre tensorielle).

. $S' \multimap T$ correspond aux fonctions *linéaires* de S' dans T, essentiellement les fonctions qui appellent leur argument exactement une fois.

Sous-jacente à la logique intuitionniste existe une *logique linéaire* (Girard, [14]), plus proche des réalités opérationnelles : elle permet de contrôler au moyen du typage les questions de stockage/lecture (voir par exemple le travail de Lafont, [15]). Surtout, elle rétablit sans inconvénients la symétrie de la logique classique, au moyen de la *négation linéaire* $S^{\perp}$ (analogue de l'espace dual); c'est ainsi que $S \multimap T$ et $T^{\perp} \multimap S^{\perp}$ sont isomorphes. Le système F se retrouve ainsi remplacé par un système plus fin, essentiellement parallèle, et dont les schémas se trouvent repartis sur cinq niveaux :

i) *communication* : la négation linéaire, qui échange entrées et sorties.

ii) *coopération* : les connecteurs ⊗, ⅋, $\multimap$, qui expriment diverses modalités de répartition d'une même tâche entre plusieurs participants.

iii) *partage* : les connecteurs ⊕ et & qui expriment diverses modalités de choix d'une tâche particulière parmi plusieurs proposées.

iv) *stockage/lecture* : les connecteurs ! et ? qui expriment diverses modalités de

réutilisation d'une même information.

v) *abstraction* : les quantificateur $\Lambda\alpha.$, $\forall\alpha.$, qui expriment diverses modalités de passage entre général et particulier.

Pour exploiter au mieux les potentialités de la logique linéaire, il est nécessaire de mettre au point un nouveau style de syntaxe, dans laquelle les considérations géométriques soient au premier plan (alors que les syntaxes traditionnelles sont très pauvres du point de vue géométrique). La solution est connue pour les trois premiers niveaux ; par exemple les niveaux i) et ii) sont basés sur l'idée de *permutation cyclique* d'un ensemble fini. L'achèvement d'une telle syntaxe, libérée des limitations de type taxinomique de la syntaxe séquentielle, résoudrait la question essentielle de la sémantique opérationnelle. (Par exemple, on sait déjà qu'une partie de l'opérationnalité, c'est l'itération d'une permutation, voir [16]).

II.2. l'arithmétique fonctionnelle

Nous avons déjà vu (I.5.) que des algorithmes, au demeurant naturels, ne sont pas typables (avec le bon type) dans le système F. Pour ce qui est du terme de Maurey, sa typabilité dépend en fait de la solution d'une équation aux types :

$T \cong U \to \underline{ent}$ $\qquad\qquad$ $U \cong T \to \underline{ent}$

dont Krivine a montré qu'elle peut être ajoutée au système F sans problèmes. Mais il ne saurait être question de fabriquer un nouveau système pour chaque λ-terme intéressant ! Le travail en cours de Krivine, ([17]) se propose, sous le nom d'*arithmétique fonctionnelle* AF_2, de créer un système de programmes prouvés suffisamment souple, où l'on pourrait par exemple créer des types. Ce système s'organise autour de trois notions :

1. Un système équationnel d'algorithmes généraux, contenant le λ-calcul. La plupart de ces algorithmes ne sont pas destinés à l'éxécution.

2. Un système de types voisin du système F : les atomes sont de la forme Xt, où t est un terme de 1. et X l'analogue d'une variable de types; on a droit au connecteur $\to$, et à deux quantifications, ΛX (analogue de $\Lambda\alpha$) et Λx (quantification sur le niveau 1.). On écrira par exemple les entiers sous la forme

$\mathbb{N}x =_d \Lambda X.Xa \to (\Lambda z.(Xz \to Xbz) \to Xx)$, où a et b sont à définir, voir plus bas.

3. Une notion de réalisabilité, notée $p \Vdash T$, avec p : un λ-terme et T : un type, définie ainsi :

. $p \Vdash Xt$ sera défini au moyen d'une définition arbitraire, vérifiant des propriétés du genre de (CR 1)-(CR 3) (voir I.4.); ici peut s'exercer la possibilité de "créer" des types.

. $p \Vdash T \to U$ ssi $\forall q(q \Vdash T \to pq \Vdash U)$

. $p \Vdash \Lambda x.U$ ssi $\forall x(p \Vdash U)$

. $p \Vdash \Lambda X.U$ ssi $p \Vdash U$, quelque soit la définition arbitraire de $q \Vdash Xu$.

Par exemple, p ||– INt est défini de telle manière que ceci ne peut se produire que quand il existe un entier n, tel que p soit le n ième entier de Church (I.5.) et t soit $b^n a$.

L'idée est que IN va être un *type valeur* : celà veut dire que

p ||– INt ssi t ε IN et p = t. Ceci est vrai,pourvu d'effectuer le bon choix pour les constantes a et b que nous avions utilisées (a : l'entier 0 de Church, b : la fonction successeur du λ-calcul). En particulier, si f est un algorithme général qui envoie IN dans IN, on essaye de trouver un λ-terme p tel que

p ||– Λx.(INx → INfx) ; le λ-terme p sera alors la forme compilée de l'algorithme f : en effet, si n est un entier, n ||– INn, donc tn ||– INfn, d'où tn = fn. Le calcul de tn donne donc le résultat voulu, mais le détour par la réalisabilité a servi à prouver la terminaison et à mettre f sous une forme plus régulière, à savoir p. De plus la méthode pour prouver des propriétés p ||– T est on ne peut plus générale : tout simplement les mathématiques courantes !

Le système pourrait fonctionner comme un système expert, à qui l'on donnerait des théorèmes mathématiques sur la relation p ||– T, à charge pour lui de typer, ou de vérifier le typage d'un λ-terme, à partir de son stock d'informations.

II.3. problèmes d'efficacité

Les considérations sur la bonne structure des algorithmes nuisent parfois à leur efficacité ; ainsi, la fonction prédecesseur P0 = 0, Pn+1 = n, de temps d'exécution très court, devient, une fois typée, notablement plus longue : pour calculer P10, on est amené à individualiser 10 en dix bâtons, puis à le reconstruire en oubliant le premier. Ce problème se pose pour toutes les données courantes (par exemple F ne sait pas enlever rapidement la dernière composante d'une liste). Récemment, Parigot (communication privée, 1987) a fait remarquer qu'une petite modification de la définition de p ||– A dans le cas de Λx., amenait à une notion voisine d'entier (n+1 = λx.λf.fn(nxf)) d'où ce type de problèmes serait exclu. Il s'agit de la première indication selon laquelle ce problème important pourrait avoir une solution élégante ; une indication seulement, car, si le problème du prédecesseur est bien résolu, la taille exponentielle de la nouvelle représentation de n va à l'encontre de l'idée d'efficacité à l'origine de ce type de problèmes.

BIBLIOGRAPHIE

[1] J.Y. GIRARD : *Une extension de l'interprétation de Gödel à l'analyse et son application à l'élimination des coupures dans l'analyse et la théorie des types*, Proceedings Second Scandinavian Logic Symp. ed. Fenstad, pp. 63-92, North Holland, Amsterdam 1971.

[2] J.Y. GIRARD : *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*, Thèse de Doctorat d'Etat, Université Paris VII,1972.
[3] P. MARTIN LØF : *Intuitionistic type theory*, Bibliopolis, Napoli, 1984.
[4] T. COQUAND, G. HUET : *A theory of constructions*, Comptes-Rendus du Congrès de Logique d'Orsay, à paraître chez North-Holland.
[5] J. REYNOLDS : *Towards a theory of type structure*, Lectures Notes in Computer Science 19, pp. 408-423, Springer-Verlag 1974.
[6] J. REYNOLDS : *Polymorphism is not set-theoretic*, Lecture Notes in Computer Science 173, pp. 145-156, Springer-Verlag 1984.
[7] D. PRAWITZ : *Natural Deduction*, Almqvitz & Wiksell, Stockholm, 1965.
[8] W. HOWARD : *The formulas-as-types notion of construction*, in To H.B. Curry : Essays on Combinatory Logic, Lambda-Calculus and Formalism, ed. Seldin et Hindley, Academic Press 1980.
[9] W.TAIT : *Intensional interpretation of functionals of finite types I*, Journal of Symbolic Logic 32 (1967), pp. 198-212.
[10] J.L. KRIVINE : *Un algorithme non typable dans le système F*, note aux CRAS, 1987.
[11] D. SCOTT : *Domains for denotational semantics*, Lecture Notes in Computer Science 140, pp. 577-613, Springer-Verlag 1982.
[12] G. BERRY : *Modèles complètement adéquats et stables des lambda-calculs typés*, Thèse de Doctorat d'Etat, Université Paris VII, 1979.
[13] J.Y. GIRARD : *The system F of variable types, fifteen years later*, Theoretical Computer Science 45 (1986), pp. 159-192.
[14] J.Y. GIRARD : *Linear Logic*, à paraître dans Theoretical Computer Science.
[15] J.Y. GIRARD, Y. LAFONT : *Linear Logic and lazy evaluation*, à paraître dans les comptes-rendus du Congrès TAPTSOFT '87, Pisa.
[16] J.Y. GIRARD : *Multiplicatives*, à paraître dans les comptes-rendus du Congrès tenu au I.S.I. de Torino, Octobre 1986.
[17] J.L. KRIVINE : *Une approche model-théorique à la programmation typée*, en cours de rédaction.

Jean-Yves GIRARD

Université de Paris 7
Equipe de Logique Mathématique
UA 753 du CNRS
T. 45-55 - 5e étage
2 place Jussieu
F-75251 PARIS CEDEX 05