Le calcul fonctionnel et ses applications à la notation mathématique, aux algorithmes informatiques, aux preuves de la logique et à la modélisation linguistique

Gérard Huet

Séminaire Collège de France, 2 décembre 2009

Introduction

Le cours a présenté jusqu'ici un formalisme important, le λ-calcul. Nous allons maintenant présenter quelques applications importantes de ce formalisme.

Une difficulté importante de compréhension du λ-calcul est sa simplicité, d'une part, et la non-familiarité de sa notation, même pour ceux qui sont familiers avec des notations mathématiques telles que $\int sin(2x)dx$, $\{x|x > 0\}$ ou encore $\forall x \in \mathbb{N} \cdot x + 1 > x$.

Il faut reconnaître que la notation ésotérique $\lambda x.(sin((\times\ 2)\ x))$ n'a pas la clarté de son équivalent traditionnel $x \mapsto sin(2x)$. Mais son universalité comme *notation fonctionnelle* la rend apte à apparaître comme dénotant une valeur mathématique fonctionnelle, et donc à débarrasser l'intégrale des scories historiques telles que dx.

Introduction (fin)

On pourra donc écrire toutes ces notations traditionnelles sous une forme homogène, en reconnaissant la fonctionnelle sous-jacente à la notation, pour simplement écrire $(Int\ \lambda x \cdot (sin\ ((\times\ 2)\ x)))$ ou mieux encore $(\lambda f \cdot \lambda x \cdot (sin\ ((\times\ 2)\ x))\ (Int\ f))$, notation qu'il est facile de linéariser en vernaculaire mathématique, par exemple *Considérons l'intégrale de* $x \mapsto sin(2x)$ ou bien encore *Soit* $f(x) = sin(2x)$ *dans* $\int f$.

L'important, c'est de faire passer le symbole $\mapsto$ du rang de paraphrase dans le *vernaculaire mathématique* à celui de notion utilisable comme sous-formule. Faut-il pour autant utiliser λ à sa place ? Peut-être la notation $[x](f\ x)$ est-elle supérieure à $\lambda x \cdot (f\ x)$? D'autres suggèrent $[x]\langle x\ f\rangle$. L'important, c'est de reconnaître l'importance d'un *calcul fonctionnel* universel.

Notation mathématique

- la fonction en x de valeur $sin(2x)$
- $x \mapsto sin(2x)$
- $\int sin(2x)dx \quad \star \int (x \mapsto sin(2x))$
- $\star(x \mapsto sin(2x)) \circ (x \mapsto x^2)$
- $\star x \mapsto (y \mapsto sin(x+y))$
- $\forall \quad \partial \quad \sum$
- $\{x \mid P(x)\}$
- assemblages de Bourbaki

λ-notation

L'idée est de prendre au sérieux la notation mathématique, avec une gestion précise des environnements dans lesquels on spécifie des axiomes, des hypothèses, des définitions, etc. La notation centrale devient la λ-notation d'Alonzo Church (1936).

Cette notation, et le λ-calcul qui l'anime, ont d'abord été utilisés pour la définition de fonctions calculables. Avant de parler du calcul, voyons d'abord la notation des λ-termes.

C'est une algèbre à 3 constructeurs très simple :

- x
- $(M\ N)$
- $\lambda x \cdot E$

Pour expliquer en quoi le λ est un *opérateur de liaison*, il convient d'abstraire de cette notation le nom des variables.

Les termes fonctionnels

On abstrait les variables au profit d'une notation relative au sein de termes fonctionnels, qui sont des arbres orientés avec 2 sortes de branchements, d'arité respective 1 et 2. Les feuilles de ces arbres sont des entiers naturels dans $\mathbb{N} = \{0, 1, 2, ...\}$.

On appelle *abstractions* les branchements unaires et *applications* les branchements binaires. Les entiers sont des *variables relatives* (ou indices de de Bruijn). Chaque variable est relative à son abstraction de liaison, car elle dénote sa profondeur dans l'emboîtement des abstractions.

On considère les termes fonctionnels dans un environnement, qui dans un premier temps se réduit à une profondeur dans $\mathbb{N}$. Les termes fonctionnels sont construits dans un environnement de profondeur n, comme suit.

Les termes fonctionnels (suite)

$$n+1 \vdash n$$

$$n+1 \vdash M \Rightarrow n \vdash \Lambda M$$

$$n \vdash M \wedge n \vdash N \Rightarrow n \vdash (M\,N)$$

Les *termes fermés* ou *combinateurs* sont constructibles à profondeur 0. Par exemple, $I = \Lambda 0$, et donc $\Delta = (I\ I) = (\Lambda 0\ \Lambda 0)$, $\Delta = \Lambda(0\ 0)$, $\Omega = (\Delta\ \Delta)$, etc. Dans un terme fermé, toutes les variables dénotent. Les *termes ouverts* ont des variables libres. Les termes (ouverts et fermés) forment l'*algèbre fonctionnelle libre* engendrée par $\mathbb{N}$.

Perlis' law

Perlis' law Someone's free variable is someone else's bound variable.

Autrement dit, les variables libres d'un terme sont les variables liées dans son contexte. Sans cette précaution, on a une notation ambiguë.

La fonction successeur

On définit le terme $Succ^k\ E$, pour E un terme et $k \geq 0$, par : $Succ^k\ E = Succ_0^k\ E$, avec $Succ_n^k$ défini par :

$$Succ_n^k\ (M\ N) = (Succ_n^k\ M\ Succ_n^k\ N)$$

$$Succ_n^k\ \Lambda M = \Lambda(Succ_{n+1}^k\ M)$$

$$Succ_n^k\ i = i \quad (i < n)$$

$$Succ_n^k\ i = i + k \quad (i \geq n)$$

La fonction de substitution

Si T est un terme, on définit le terme $Subst\ T\ E = Subst_0^T\ E$ par :

$$Subst_n^T\ (M\ N) = (Subst_n^T\ M\ Subst_n^T\ N)$$

$$Subst_n^T\ \Lambda M = \Lambda(Subst_{n+1}^T\ M)$$

$$Subst_n^T\ i = i \quad (i < n)$$

$$Subst_n^T\ i = Succ^n\ T \quad (i = n)$$

$$Subst_n^T\ i = i - 1 \quad (i > n)$$

Théorème de Substitution

Théorème.

$$M[x \leftarrow N][y \leftarrow P] = M[y \leftarrow P][x \leftarrow N[y \leftarrow P]]$$

ou, plus précisément :

$$Subst_n^P\ (Subst\ N\ M) = Subst\ (Subst_n^P\ N)(Subst_{n+1}^P\ M)$$

Ce théorème est la clé du pavage des dérivations par les résidus, et donc de la confluence du calcul fonctionnel qui suit.

Pour en savoir plus :
Constructive Computation Theory. Course notes on λ-calculus. Voir `http://yquem.inria.fr/~huet/PUBLIC/CCT.pdf`. Ce cours est exécutable en Ocaml. Le source en est téléchargeable à l'URL `http://yquem.inria.fr/~huet/PUBLIC/LAMBDA.tar.gz`.

Le calcul fonctionnel

Les termes fonctionnels sont munis d'une relation de calcul, appelée β-réduction. C'est la congruence engendrée par :

$$(\Lambda M\ N) \rightarrow_\beta Subst\ N\ M$$

Théorème (Church et Rosser). La β-réduction est une relation confluente.

Corollaire. La forme normale d'un terme (forme irréductible obtenue par une séquence de calculs) est unique, quand elle existe.

Corollaire. Le calcul fonctionnel, bien que non-déterministe, est *déterminé*.

Le calcul fonctionnel appliqué à la notation mathématique

On peut maintenant régulariser la notation mathématique.

$$\forall x \cdot P(x) =_{def} (All\ P)$$

$$\int f(x)dx =_{def} (Int\ f)$$

$$\partial f/\partial x =_{def} (Der\ f)$$

$$x \in y =_{def} (y\ x)$$

$$\{x \mid E(x)\} =_{def} \Lambda E$$

Le paradoxe n'est pas loin

$$\{x|x \in x\} = \Lambda(0\ 0) = \Delta$$

$$\Delta \in \Delta = (\Delta\ \Delta) = \Omega$$

La non terminaison de ce dernier calcul exhibe l'incohérence de ce système de notation. Ce qui n'est pas différent de l'incohérence de la notation ensembliste dans un discours ensembliste naïf.

La solution classique pour évacuer le paradoxe du menteur est de stratifier la relation d'appartenance, et donc ici l'application.

On peut néanmoins construire des modèles du calcul fonctionnel, comme l'a montré Dana Scott. Mais il n'est pas possible de toutes façons de concilier cohérence logique et complétude combinatoire, car la négation ne possède pas de point fixe sans incohérence.

Historique ancien

L'inventeur du calcul fonctionnel est Gottlob Frege (1879), sous le nom de *Begriffsschrift* (langage des formules pour la pensée pure, ou calcul conceptuel). Il l'a utilisé pour définir un calcul des prédicats non restreint au premier ordre, qui était incohérent.

Whitehead et Russell ont utilisé une version d'un λ-calcul restreint par un système de types compliqué dans leur monument "Principia Mathematica" (1907). Leurs formules utilisaient des variables liées notées $\hat{\mathrm{x}}$, et l'accent circonflexe a été linéarisé dans la formule en λ par Church, qui a le premier exhibé le calcul fonctionnel sous le nom de λ-calcul et prouvé ses principales propriétés en 1936.

Il tenta alors de fonder une logique sur ce calcul, tentative montrée erronée par Rosser. Il se résigna à stratifier les termes d'un calcul fonctionnel restreint par des types, pour proposer une "Théorie des types simples" pour formaliser les mathématiques (1940).

Les types simples

Les types simples sont la fermeture par la flèche fonctionnelle d'un ensemble fini de types atomiques $A = \{\iota, o, ...\}$. Le type $\alpha \to \beta$ dénote l'espace des fonctions de domaine α et de codomaine β.

On restreint maintenant le calcul fonctionnel en limitant les règles de construction des termes. On peuple les contextes avec des types, comme suit.

$$\gamma_n \times ... \times \gamma_0 \vdash k : \gamma_k$$

$$\Gamma \vdash M : \alpha \to \beta \wedge \Gamma \vdash N : \alpha \Rightarrow \Gamma \vdash (M\ N) : \beta$$

$$\Gamma \times \alpha \vdash M : \beta \Rightarrow \Gamma \vdash \Lambda M : \alpha \to \beta$$

Ces jugements de typage permettent de définir la relation "M admet le type α" par $\emptyset \vdash M : \alpha$ pour un terme fermé M. Ainsi, I admet tout type de la forme $\alpha \to \alpha$, et Δ n'est pas typable.

Les types simples (suite)

Cette notion de typage est due à Curry. Une autre méthode, due à Church, incorpore la notion de type aux termes, en remplaçant l'abstraction ΛM par un lieur explicitement typé $[\tau]$. Le typage devient alors une fonction, qui associe à tout terme explicitement typé et typable un type unique.

$$t_\Gamma(n) = \Gamma_n$$

$$t_\Gamma([\tau]M) = \tau \to t_{\Gamma\times\tau}(M)$$

$$t_\Gamma((M\ N)) = \tau \Leftrightarrow t_\Gamma(M) = t_\Gamma(N) \to \tau$$

et on définit $type(M) = t_\emptyset(M)$.

Les types simples (suite)

Ajoutons des variables formelles aux types pour typer maintenant avec des schémas de types comme $\alpha \to \alpha$. On dit qu'un type τ est *plus général* qu'un type ρ, noté $\tau \leq \rho$, ssi ρ est une instance de τ par une substitution associant un type à toute variable de type.

Théorème d'Hindley. Tout terme typable possède un *type principal*, minimal au sens de l'ordre partiel de généralité.

Par exemple, $I = [x]x$ a pour type principal $\alpha \to \alpha$, $K = [x, y]x$ a pour type principal $\alpha \to (\beta \to \alpha)$, $S = [x, y, z]((x\ z)\ (y\ z)))$ a pour type principal $(\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))$

Les types simples (fin)

Théorème (Tait-Gandy). Tout terme typable est *fortement normalisable*, c'est-à-dire que toute séquence de réductions termine (sur sa forme normale).

La théorie des types simples de Church

On prend o pour le type des valeurs de vérités $\{Vrai, Faux\}$, et ι pour le type des entiers $\mathbb{N}$. On axiomatise $0 : \iota$, $S : \iota \to \iota$. On se donne des connectives logiques comme $\vee : o \to (o \to o)$ et $\forall_\alpha : (\alpha \to o) \to o$. On obtient ainsi un calcul des prédicats d'ordre supérieur, qu'on peut munir d'une logique. Le calcul fonctionnel permet d'écrire facilement le principe de récurrence, non plus sous forme d'un schéma d'axiome, mais comme axiome unique : $\forall(\lambda P.(P\ 0) \Rightarrow (\forall(\lambda n.((P\ n) \Rightarrow (P\ (S\ n)))) \Rightarrow \forall P))$, avec $P : \iota \to o$.

mais aussi l'égalité d'indiscernabilité, due à Leibniz :

$$x = y =_{def} \forall(\lambda P.(P\ x) \Rightarrow (P\ y))$$

La théorie des types simples de Church

Cette théorie des types est préférable à la logique de premier ordre par son expressivité, mais ses propriétés méta-théoriques sont similaires. Notamment, le théorème de complétude est valide, pourvu que les espaces fonctionnels interprétant le type $\alpha \rightarrow \beta$ ne soient pas restreints à l'ensemble de toutes les fonctions de l'ensemble modèle de α vers l'ensemble modèle de β. De même, la théorie des types simples n'est pas complète, et admet des propositions indécidables, comme par exemple la proposition qui internalise "je ne suis pas démontrable", ou encore celle qui exprime la cohérence du système logique.

Cette théorie a donné lieu à trois implémentations informatiques :
- G. Huet Constrained resolution with ho-unification (1972)
- M. Gordon, R. Milner, C. Wadsworth LCF (1979)
- M. Gordon HOL (1985)

Les types interprétés par des propositions logiques

Réexprimons les règles de typage comme règles d'inférence logique :

$$\Gamma \vdash x : \tau \quad ([x : \tau] \in \Gamma)$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (M\ N) : \tau}$$

$$\frac{\Gamma[x : \sigma] \vdash M : \tau}{\Gamma \vdash \lambda x \cdot M : \sigma \to \tau}$$

En particulier, la règle de typage de l'application, quand on oublie le terme fonctionnel, s'écrit :

$$\frac{\Gamma \vdash \sigma \to \tau \quad \Gamma \vdash \sigma}{\Gamma \vdash \tau}$$

On reconnaît la règle logique de *Modus Ponens*. En fait ces règles définissent les règles d'inférence d'une logique minimale, dont les variables propositionnelles sont les variables de types, la seule connective étant l'implication, notée par la flèche de fonctionnalité.

Des fonctions aux preuves

$$\Gamma \vdash n : \Gamma_n$$

$$\frac{\Gamma \vdash M : \sigma \Rightarrow \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (M\ N) : \tau}$$

$$\frac{\Gamma \times \sigma \vdash M : \tau}{\Gamma \vdash \Lambda M : \sigma \Rightarrow \tau}$$

Autrement dit, les règles de typage valident des procédés de preuve, respectivement la règle axiome pour les variables, l'introduction de $\Rightarrow$ pour l'abstraction et son élimination pour l'application. Les termes fonctionnels typés sont ainsi vus comme des justifications des propositions logiques que sont leurs types.

La règle de calcul de β-réduction devient alors un processus d'explicitation des arguments logiques, où les définitions et les lemmes sont expansés. On parle d'*élimination des coupures*.

Isomorphisme de Curry, Howard et de Bruijn

Le calcul fonctionnel typé est isomorphe à une logique constructive présentée sous la forme de *déduction naturelle* (Gentzen 1936). Les types sont les propositions, et les termes sont les preuves.

Cet isomorphisme a d'abord été utilisé par Curry (1958), dans une théorie des combinateurs isomorphe à la présentation du calcul implicationnel minimal à la Hilbert. L'opérateur d'abstraction remplace avantageusement les combinateurs S et K, d'emploi difficile. D'ailleurs, la preuve du théorème de déduction dans le système d'Hilbert peut être vue comme l'algorithme d'abstraction dans les systèmes à combinateurs.

Types dépendants

Cet isomorphisme a ensuite été étendu au calcul des prédicats du premier ordre par Howard ($\approx$ 1970). Pour cela, on introduit un opérateur de produit dépendant sur les types, ayant les mêmes propriétés formelles que le quantificateur universel :

$$\forall x : \tau \cdot P(x) \quad \equiv \Pi[x : \tau](P\ x) \quad (P : \tau \to o)$$

Théorie de la démonstration et Calcul fonctionnel

Cet isomorphisme a longtemps été ignoré. Gentzen a anticipé le λ-calcul typé avec son système de déduction naturelle, et en a proposé une variante avec le calcul des séquents. Le théorème dit d'élimination des coupures était un théorème de terminaison des calculs (normalisation forte), permettant de montrer la cohérence du système.

Même en 1965 le livre classique de Prawitz sur la déduction naturelle ignore cette connexion. La théorie de la démonstration et le calcul fonctionnel ont été développés largement indépendamment par les logiciens et les informaticiens. Il fallut attendre 1989 avec la publication du livre "Proofs and Types" par Jean-Yves Girard, Yves Lafont et Paul Taylor, pour que les deux points de vue se rejoignent définitivement.

Calculs fonctionnels avec des types dépendants

C'est de Bruijn qui a fait l'unification fondamentale en mélangeant les idées de Church (fonctionnelles d'ordre supérieur) avec celles de Gentzen/Howard (preuves structurées avec des contextes d'hypothèse) au sein d'un système de représentation des mathématiques appelé Automath. Le calcul fonctionnel sous-jacent à Automath était typé par des types qui étaient eux-mêmes des abstractions (l'abstraction au niveau des types étant la quantification universelle). Ce système, qui a donné lieu à d'importants développements à Eindhoven dans les années 70 et 80, n'a pas eu l'impact mérité, probablement parce qu'il était trop précurseur.

Vernaculaire mathématique

En parallèle, Nicholas de Bruijn a posé les bases d'un *vernaculaire mathématique* apte à exprimer fidèlement un argumentaire rigoureux de développement mathématique. Il vise à traduire les tournures linguistiques concernant les axiomes, les définitions, les hypothèses, les conjectures et les preuves de manière non ambiguë en calcul fonctionnel typé à la Automath.

Le problème de définir un langage contrôlé pour servir d'interface entre un mathématicien et un assistant à la preuve est toujours d'actualité.

Après Automath

Dans les années 90, les idées d'Automath ont été reprises par Plotkin et Harper au sein du système formel **LF** (Logical Frameworks).

Enfin, l'ajout d'un opérateur de somme dépendante symétrique du produit dépendant permit la définition par Per Martin-Löf d'une *Théorie intuitioniste des types* proposée comme fondement constructif des mathématiques (1984).

De même, les idées d'Automath ont été déterminantes pour la conception du *Calcul des Constructions* de Coquand et Huet, comme nous le verrons plus loin. Mais nous faisons d'abord un détour par les langages de programmation.

Langages de programmation fonctionnels (préhistoire)

Le calcul fonctionnel permet une présentation particulièrement élégante de la théorie des fonctions calculables (ou fonctions partielles récursives) et de son extension aux fonctionnelles d'ordre supérieur (Church 1936, Kleene).

Son utilisation sérieuse comme langage de programmation a été proposée par Peter Landin dans un article célèbre (The next 700 programming languages, 1966). L'idée derrière le formalisme ISWIM est d'utiliser le calcul fonctionnel comme langage algorithmique générique, incorporant deux primitives de contrôle, la conditionnelle (expressions if-then-else) et la récursion (combinateur de point fixe muni d'une règle de calcul autonome plutôt que son codage en combinateur Y ou autre). Ce langage fonctionnel noyau peut être étendu pour les besoins des applications par des bibliothèques spécifiques à des structures de données.

Le mécanisme de calcul choisi par ISWIM est une variante plus faible de la β-réduction (interdisant la réduction sous les abstractions). Ce qui permet de prendre comme stratégie de calcul l'appel dit "par valeur", où les radicaux sont réduits de l'intérieur vers l'extérieur, et de la gauche vers la droite. Une syntaxe spéciale permet de lire les radicaux comme des définitions utiles à la structuration en sous-programmes. Ainsi $(\Lambda M\ N)$ peut s'écrire :

$$let\ x = N\ in\ M \quad =_{def} ([x]M\ N) \ = \quad M\ where\ x = N$$

Les idées d'ISWIM ont été reprises par Plotkin avec le langage PCF, qui sous-tend une variante de théorie des fonctions arithmétiques récursives interprétées en sémantique dénotationnelle (domaines de Scott).

Langages de programmation fonctionnels (suite)

Nous avons vu que les termes fonctionnels pouvaient être typés dans un système de typage avec des variables libres. Ceci est une première approche de polymorphisme, avec un résultat de typage principal dû à Hindley. On peut considérer le type $\alpha \rightarrow \beta$, avec α et β des variables de type, comme un type universellement quantifié $\forall\alpha\forall\beta\ \alpha \rightarrow \beta$.

On peut alors utiliser la notation *let* d'ISWIM pour étendre le typage d'un terme avec une simulation limitée du calcul.
Définissons le *type à la Milner* d'un terme ISWIM comme le type principal à la Hindley, s'il existe, du terme obtenu en faisant la β-réduction en une étape de tous les radicaux explicités par des définitions *let*. Le formalisme obtenu a été conçu par Robin Milner pour la conception du langage ML, au départ méta-langage du système de preuves de programmes LCF (1979).

On obtient ainsi un langage de programmation applicatif très puissant, où les types reflètent une cohérence des structures de données sans pour autant imposer la terminaison des programmes, l'opérateur de récursion Fix_α étant déclaré de type $(\alpha \to \alpha) \to \alpha)$. De plus, l'utilisateur n'a pas besoin de spécifier les types, car le compilateur synthétise le type principal de tout terme typable.

ML et ses successeurs

Le langage ML a donné naissance à plusieurs implémentations importantes : Standard ML et Objective Caml. Il a profondément influencé la conception du langage Haskell, qui se rapproche du calcul fonctionnel pur en autorisant la β-récursion forte, c'est-à-dire non restreinte, et la règle d'évaluation dite normale (de l'extérieur vers l'intérieur, de gauche à droite).

Enfin, le calcul fonctionnel a inspiré la conception de systèmes de modules paramétriques, tels que les foncteurs de Xavier Leroy disponibles en Objective Caml.

Systèmes de définitions sémantique

Le calcul fonctionnel, sous une forme proche de ML, a été utilisé abondammment pour la description de la sémantique dénotationnelle des langages de programmation. Cette tradition a ses racines dans la définition sémantique complète d'Algol 60 en λ-calcul par Peter Landin (1965). Dans les années 60, un calcul fonctionnel basé sur un système de types inspiré par les équations aux domaines de Scott s'est dégagé, pour exprimer récursivement la sémantique dénotationnelle d'un langage de programmation.

Un certain nombre de combinateurs génériques ont été dégagés, pour manipuler les concepts d'environnement, de mémoire ("store") et de continuation. Les travaux essentiels sont dûs à John Reynolds, Gordon Plotkin, Gilles Kahn, Gérard Berry, et beaucoup d'autres.

Fondements de la programmation

On utilise Pidgin ML comme méta-langage (macro-génération).

Notation : [x]M pour $\lambda x \cdot M$

```
(* Bool *)
value _True =  <<[x,y]x>>
and _False = <<[x,y]y>>
and _Cond = <<[p,x,y](p x y)>>;

(* Pairs *)
value _Pair = <<[x,y,p](p x y)>>
and _Fst = <<[pa](pa ^True)>>
and _Snd = <<[pa](pa ^False)>>;

(* Turing's fixpoint combinator *)
value _Fix = <<([x,f](f (x x f)) [x,f](f (x x f)))>>;
```

L'arithmétique selon Church

```
(* Nat : Church's natural numbers *)
value _Zero = <<[s,z]z>> (* same as _False *)
and _Succ = <<[n][s,z](s (n s z))>>;

(* Church *)
value church n = iter s n _Zero
  where s _C = nf<<(^Succ ^C)>>;

value _Add = <<[m,n][s,z](m s (n s z))>>;

value _Mult = <<[m,n,x](m (n x))>>;

value _Exp = <<[m,n](n m)>>;
```

Lisp en λ-calcul

```
value _Nil = <<[c,n]n>>      (* same as _Zero *)
and _Cons = <<[x,l][c,n](c x (l c n))>>;

(* list : int list -> term *)
value rec list = fun
  [ [x::l] -> let _Cx = church x and _Ll = list l
              in <<[c,n](c ^Cx (^Ll c n))>>
  | []   -> _Nil
  ];

(* Append *)
value _Append = <<[l,l'][c,n](l c (l' c n))>>;
```

Quicksort en λ-calcul

```
value _Quicksort =
  <<(^Fix [q]let sort = [a,l]
                let p = (^Partition (^Geq a) l) in
                (^Append (q (^Fst p)) (^Cons a (q (^Snd p))))
              in [l](l sort ^Nil))>>;

let _L=list[3;2;5;1] in normal_list<<(^Quicksort ^L)>>;
 = [1; 2; 3; 5]  (27s)

let _L=list[3;2;5;1] in applicative_list<<(^Quicksort ^L)>>;
 = [1; 2; 3; 5]  (2s)
```

Le terme Quicksort comme λ-term [assembleur]

```
_Quicksort;
- : Term.term =
([x0,x1](x1 (x0 x0 x1)) [x0,x1](x1 (x0 x0 x1)) [x0]([x1,x2]
(x2 x1 [x3,x4]x4) [x1,x2]([x3]([x4,x5,x6,x7] (x4 x6 (x5 x6 x7))
(x0 ([x4] (x4 [x5,x6]x5) x3)) ([x4,x5,x6,x7](x6 x4 (x5 x6 x7))
x1 (x0 ([x4](x4 [x5,x6]x6) x3)))) ([x3,x4]([x5](x4 x5 ([x6,x7,x8]
(x8 x6 x7) [x6,x7]x7 [x6,x7]x7)) [x5,x6]([x7]([x8](x3 x5
([x9,x10,x11](x11 x9 x10) ([x9,x10,x11,x12](x11 x9 (x10 x11 x12))
x5 x7) x8) ([x9,x10,x11](x11 x9 x10) x7 ([x9,x10,x11,x12](x11 x9
(x10 x11 x12)) x5 x8))) ([x8](x8 [x9,x10]x10) x6)) ([x7](x7
[x8,x9]x8) x6))) ([x3,x4](x3 ([x5,x6]([x7](x7 [x8,x9]x9) (x6 x5
([x7,x8,x9](x9 x7 x8) [x7,x8]x8 [x7,x8]x8))) [x5]([x6]([x7,x8,x9]
(x9 x7 x8) ([x7,x8,x9](x8 (x7 x8 x9)) x6) x6) ([x6](x6 [x7,x8]x7)
x5))) x4 ([x5,x6]x5 [x5,x6]x6) [x5,x6]x5) x1) x2))))
```

Quicksort comme terme fonctionnel [langage machine]

Avec une fonction d'impression de plus bas niveau :

```
_Quicksort;
- : Term.term =
(^^(0 (1 1 0)) ^^(0 (1 1 0)) ^(^^(0 1 ^^0) ^^(^(^^^^(3 1 (2 1 0))
(3 (^(0 ^^1) 0)) (^^^^(1 3 (2 1 0)) 2 (3 (^(0 ^^0) 0)))) (^^(^(1
0 (^^^(0 2 1) ^^0 ^^0)) ^^(^(^(5 3 (^^^(0 2 1) (^^^^(1 3 (2 1 0))
3 1) 0) (^^^(0 2 1) 1 (^^^^(1 3 (2 1 0)) 3 0))) (^(0 ^^0) 1)) (^(0
^^1) 0))) (^^(1 (^^(^(0 ^^0) (0 1 (^^^(0 2 1) ^^0 ^^0))) ^(^(^^^(0
2 1) (^^^(1 (2 1 0)) 0) 0) (^(0 ^^1) 0))) 0 (^^1 ^^0) ^^1) 1) 0))))
```

Question : Quelle est la machine ? Réponses : Landin, Krivine,
G-machine, F-machine, CAM, substitutions explicites, Gallium, etc.

Le calcul fonctionnel polymorphe

Nous avons vu que ML était muni de types ayant un certain niveau de polymorphisme, exprimé par la quantification universelle sur les variables de termes. Mais cette quantification est restreinte à une forme prénexe, et n'autorise pas l'utilisation de types quantifiés en position contravariante (à gauche des flèches). Si l'on permet cette possibilité, on obtient un calcul fonctionnel typé beaucoup plus puissant, appelé **calcul fonctionnel polymorphe**. Girard a défini ce calcul (système **F**) en 1970, et a prouvé sa cohérence, résolvant une conjecture logique célèbre.

Pour comprendre la genèse de ce formalisme, il faut remonter aux entiers de Church (1936). C'est en fait la première véritable application du calcul fonctionnel, pour développer une théorie des fonctions partielles calculables. Le λ-calcul donne un moyen plus direct d'exprimer les fonctions récursives partielles que les systèmes de définitions arithmétiques récursives de Kleene ou autres.

Les entiers de Church

Church a imaginé le codage suivant d'un entier en calcul fonctionnel. L'entier n est codé par $[s][z](s\ (s...\ (s\ z)))$ avec n étages de s. En calcul fonctionnel, $\Lambda\Lambda(1\ (1...\ (1\ 0)))$. Vous voyez, c'est l'entier en notation unaire. Mais c'est un entier doublement fonctionnel, qui peut effectivement itérer une fonction $S : nat \to nat$ à partir de $0 : int$ pour former $((n\ S)\ 0) \to^*_\beta S(S(...S(0))) : nat$, mais qui peut plus généralement itérer une fonction $f : \alpha \to \alpha$ à partir d'une valeur initiale $x : \alpha$, et ceci pour tout type α. L'idée est donc d'exprimer le polymorphisme de tels entiers fonctionnels par le type

$$nat = \forall\alpha \cdot (\alpha \to \alpha) \to (\alpha \to \alpha)$$

Des structures de données comme structures de contrôle

Regardons dans les yeux le type des entiers polymorphes :

$$nat = \forall\alpha \cdot (\alpha \to \alpha) \to (\alpha \to \alpha)$$

Cette écriture reflète bien la nature d'itérateur d'un tel entier polymorphe analogue à la boucle `for i=1 to n do {...}` où $s : \sigma \to \sigma$ est la fonction de transition d'états de `{...}` et $z : \sigma$ l'état initial de la machine.

Plus généralement, on code une structure de données comme une structure de contrôle en attente de ses arguments (ses méthodes, comme on dirait en programmation objet).

L'arithmétique de Church

On a maintenant $Zero = \Lambda\Lambda 0$ avec $Zero : nat$. En effet, on vérifie :

$$[s : nat \to nat][z : nat]z : nat$$

Il faut maintenant programmmer la fonction successeur. L'intuition est de faire un tour de boucle supplémentaire au successeur générique s. On obtient alors $Succ = [n][s, z](s\ (n\ s\ z))$ que vous préférez sans doute à sa valeur formelle $\Lambda\Lambda\Lambda(1\ ((2\ 1)\ 0))$.

Mais pour typer $Succ : nat \to nat$ nous avons deux difficultés. La première est d'instancier le type universellement quantifié nat de l'argument n pour un type particulier α correspondant à ses arguments suivants $s : \alpha \to \alpha$ et $z : \alpha$. La deuxième est de généraliser le type du résultat pour transformer $(\alpha \to \alpha) \to (\alpha \to \alpha)$ en sa généralisation nat.

Coercions de type

Grâce à l'annotation du terme fonctionnel avec ces deux opérations de coercion de type, on peut expliciter :

$$Succ = [n : nat]\forall\alpha.[s : \alpha \to \alpha][z : \alpha](s\ (\langle n\ \alpha\rangle\ s\ z))$$

On voit donc apparaître deux annotations des termes fonctionnels par des types universellement quantifiés :

– $\Gamma \times type(\alpha) \vdash M : \tau \Rightarrow \Gamma \vdash \forall\alpha.M : \forall\alpha.\tau$

– $\Gamma \vdash M : \forall\alpha.\tau \wedge \Gamma \vdash \sigma : type \Rightarrow \Gamma \vdash \langle M\ \sigma\rangle : \tau[\alpha \leftarrow \sigma]$

On complète le système par la définition de $\Gamma \vdash \sigma : type$ par les 3 règles de formation de types (axiome, $\to$, $\forall$) et on obtient ainsi le système **F** de Girard.

L'arithmétique de Church (suite)

On peut maintenant itérer le successeur pour obtenir l'addition :

$$Add = [n : nat][m : nat][s : \alpha \to \alpha][z : \alpha](n\ s\ (m\ s\ z))$$

Il s'agit d'un algorithme d'addition parmi d'autres, on pourrait commuter n et m par exemple.

Après, on obtient la multiplication, par exemple

$$Mult = [n : nat][m : nat][s : \alpha \to \alpha](n\ (m\ s))$$

qui s'amuse à composer deux entiers comme deux boucles emboîtées.

De plus en plus fort, on peut programmer l'exponentielle par simple application :

$$Exp = [n : nat][m : nat](m\ n)$$

L'arithmétique de Church (suite)

Chacun de ces exemples est typable avec les types explicitement quantifiés, par utilisation appropriée des décorations de généralisation et d'instanciation, munies des règles logiques de la quantification universelle.

Par contre, bien évidemment, l'exponentiation n'est pas typable avec les types simples, ni même (correctement) avec le polymorphisme prénexe de ML (Exercice). D'où l'interrogation : "Pourquoi Church a-t-il formulé une théorie des types qui interdise l'arithmétique de Church ?".

Il y a tout de même là un paradoxe - il n'y a pas confluence dans le λ-calcul entre les entiers de Church et la théorie des types de Church !

L'arithmétique de Church sauvée par Girard

La réponse est historique. Church en 1936 a inventé le λ-calcul comme support des fonctions partielles récursives avec le codage des entiers polymorphes, mais le système de typage correspondant n'a été établi, avec sa propriété de normalisation forte, qu'en 1970 par Girard avec le système **F** de calcul fonctionnel polymorphe. En 1940, lorsque Church a construit sa logique d'ordre supérieur, il a dû brider l'expressivité du calcul par des types pour éviter l'incohérence, et en se limitant aux types simples il a dû renoncer à internaliser l'arithmétique pour la laisser sous forme axiomatique.

Nous savons maintenant qu'on peut avoir le beurre (la cohérence) et l'argent du beurre (l'arithmétique de second ordre, c'est-à-dire l'analyse).

Puissance du polymorphisme général

En ML :

```
# value distr_pair f x y z = z (f x) (f y);
```

$$distr\_pair : (\alpha \to \beta) \to \alpha \to \alpha \to (\beta \to \beta \to \gamma) \to \gamma$$

L'unification des types de x et y en α est une restriction inutile. Dans le calcul fonctionnel polymorphe on obtient la généralité voulue :

$$distr\_pair : (\forall\alpha.(\alpha \to \alpha)) \to \alpha \to \beta \to (\alpha \to \beta \to \gamma) \to \gamma$$

Puissance du polymorphisme général (suite)

Nous avons vu que la quantification en position contravariante permettait de typer uniformément l'arithmétique de second ordre et des algorithmes généraux comme $distr\_pair$. Mieux :

```
# value delta x = x x;
Error: ill-typed
```

$$\Delta : \mathbf{1} \to \mathbf{1}$$

avec $\Delta = [x : \mathbf{1}](\langle x\ \mathbf{1}\rangle\ x)$ où $\mathbf{1}{=}\forall\alpha.\alpha \to \alpha$ est le type de l'égalité polymorphe $I = \forall\alpha.[x : \alpha]x$.

Mais $\mathbf{0}{=}\forall\alpha.\alpha$ est vide, et Ω n'est pas typable.

F^ω

On peut aller plus loin dans le polymorphisme, et autoriser de quantifier non seulement sur des types, mais sur des opérateurs de types. On a alors l'équivalent du calcul fonctionnel simplement typé, appliqué à la structure des types d'un calcul formidablement polymorphe. Le système correspondant F^ω est défini, et sa cohérence est établie, dans la thèse de Jean-Yves Girard (1970).

Un système similaire au système F a été proposé indépendamment par John Reynolds en 1974, comme langage de programmation polymorphe, la généralisation correspondant à F^ω a été proposée par Nancy McCracken en 1979.

Programmation fonctionnelle algébrique

Ce n'est que dans les années 80 qu'on a compris la méthodologie générale de programmation sur des types algébriques codés en calcul fonctionnel polymorphe, expliquée par Corrado Böhm et Alessandro Berarducci (Automatic Synthesis of typed Lambda-programs on Term Algebras, 1985).

On peut donc maintenant comprendre les programmes comme Quicksort fonctionnel ci-dessus. A la signature de l'algèbre des listes

$$list(\alpha) = [Cons : \alpha \times list(\alpha)][Nil : list(\alpha)]$$

correspond le type du système **F** :

$$\forall\beta.(\alpha \to (\beta \to \beta)) \to (\beta \to \beta)$$

et on peut maintenant comprendre les listes comme des itérateurs de séquences polymorphes, et plus généralement les structures de données comme des fonctionnelles partiellement appliquées.

Plus récemment

Il est possible de concilier polymorphisme et types dépendants. Le Calcul des Constructions de Coquand et Huet est un calcul fonctionnel polymorphe permettant à la fois les structures d'Automath et le polymorphisme sur des opérateurs de type de F^{ω} pour la structure logique. Sa cohérence repose sur la normalisation forte du calcul ainsi défini, établie dans la thèse de Thierry Coquand (1985).

Le codage des structures de données par des objets fonctionnels n'est pas entièrement satisfaisant, car des termes parasites viennent peupler les types, comme cela a été montré par Christine Paulin en 1989. Il est plus satisfaisant d'incorporer au système de types une notion primitive de types et de prédicats inductifs pour développer l'algorithmique. Le Calcul des Constructions Inductives, qui sous-tend le système Coq, bénéficie de ces derniers développements.

PTS et zoologie des systèmes de types

On a maintenant une problématique générale de théorie des types, qui peuvent être classifiés au sein d'une famille de systèmes de types purs (PTS). Ainsi le Cube de Barendregt représente le treillis des 3 extensions possibles du système des types simples par le polymorphisme, les opérateurs de type et les types dépendants.

Théorie des Types appliquée aux Mathématiques

La théorie des types est encore un sujet de recherches intensives. Son développement est motivé par l'enjeu industriel de la certification de logiciels et protocoles de réseau dont toute notre économie dépend. De plus, l'isomorphisme de Curry-Howard permet d'extraire des preuves constructives des algorithmes formellement spécifiés. La programmation passe ainsi d'une activité artisanale à une production scientifique et mathématique.

Ainsi, la preuve récente par Gonthier du théorème des 4 couleurs en Coq montre que la technologie de preuves formelles est au niveau des mathématiques contemporaines les plus profondes.

L'avantage du polymorphisme restreint de ML est qu'il permet la synthèse de types, puisqu'on peut calculer le type principal d'une terme typable, sans avoir besoin pour le programmeur d'annoter ses algorithmes par des types. Ce polymorphisme permet le partage de bibliothèques, par exemple de manipulation de listes génériques.

Le système de types polymorphes de F est nécessaire pour faire les codages arithmétiques ci-dessus. Il est en effet essentiel de pouvoir quantifier une sous expression de type en position contravariante (à gauche d'une flèche), comme pour nat. Mais il a l'inconvénient de ne pas définir de type principal. Ainsi, $Zero = \Lambda\Lambda 0$ en tant que terme fonctionnel est identique à $False : bool$ avec $bool = \forall\alpha \cdot \alpha \to (\alpha \to \alpha)$. On est donc obligé d'annoter les termes avec un minimum de typage explicite pour spécifier leur utilisation de manière vérifiable, car la F-typabilité est indécidable.

Vers les langages de programmation du futur

L'extension du langage ML par un polymorphisme général est l'objet d'actives recherches, notamment autour d'Objective Caml (MLF).

Le langage Haskell est une démonstration éclatante de la pertinence du calcul fonctionnel pur pour le développement de logiciels robustes. Il compense l'absence de modules par un système de classes très élégant.

L'utilisation de types dépendants en programmation est encore dans ses balbutiements (Cayenne). Néanmoins, on peut s'attendre à ce que les systèmes de preuve comme Coq fournissent l'ossature de langages de programmation puissants munis de spécifications logiques de très haut niveau.

Types conjonctifs

Revenons au système d'inférence de types simples :

$$\Gamma \vdash x : \tau \quad ([x : \tau] \in \Gamma)$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (M\ N) : \tau}$$

$$\frac{\Gamma[x : \sigma] \vdash M : \tau}{\Gamma \vdash \lambda x \cdot M : \sigma \to \tau}$$

Ce système impose à toutes les occurrences de la variable x d'avoir des types homogènes. Si on relaxe cette exigence, on obtient un système plus souple, où les lieurs typent leur variable formelle avec une conjonction $\tau_1 \wedge \tau_2 \wedge ...\tau_n$. En notant $\sigma \leq \sigma' \Leftrightarrow \exists \tau . \sigma \equiv \sigma' \wedge \tau$ on obtient une relation de sous-typage, qui permet de typer de façon plus générale les applications :

$$\frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma' \quad \sigma \leq \sigma'}{\Gamma \vdash (M\ N) : \tau}$$

Types conjonctifs (fin)

Cette méthode de typage par des types conjonctifs est due à Mario Coppo and Mariangiola Dezani-Ciancaglini (1980). Il en existe deux variantes, les systèmes **D** et **E**. Le système **D** (resp. **E**) caractérise les termes fonctionnels fortement normalisables (resp. normalisables). En conséquence, ni **D** ni **E** ne sont décidables, et l'intérêt de ces systèmes est donc relativement théorique.

Ces systèmes, et les preuves de leur caractérisation de la terminaison, sont détaillés dans la monographie de Jean-Louis Krivine "Lamba-calcul – types et modèles" (1990).

Résumé

Le **Calcul Fonctionnel** est un cadre puissant pour développer les mathématiques et programmer des algorithmes certifiés dans le cadre de théories des types variées. Dans sa version pure, il fournit un mécanisme de calcul complet au sens de Turing.

Nous allons maintenant évoquer rapidement son utilisation en linguistique formelle.

On symétrise la flèche

$A\backslash B$ le type des fonctions prenant leur argument $(b : B)$ à gauche
A/B le type des fonctions prenant leur argument $(b : B)$ à droite.

$$Jean\ aime\ Marie.$$

$$aime : (NP\backslash S)/NP$$
$$Jean : NP$$
$$Marie : NP$$

$$aime\ Marie : NP\backslash S$$
$$Jean\ aime\ Marie : S$$

Les termes de ce λ-calcul symétrisé sont les arbres d'analyse de la syntaxe, les types sont les catégories syntaxiques. Les λ-termes sont restreints par une condition de linéarité. Ce sont les grammaires catégorielles (Lambek 1954), avec 30 ans d'avance sur la logique linéaire non commutative.

Après la syntaxe, la sémantique

Les *grammaires de Montague* (1974) sont l'expression de la sémantique des langues naturelles où le contenu logique d'une phrase est représenté dans la théorie des types de Church avec trois sortes. En plus des propositions et des objets dénotés par le discours, un type des états permet de modéliser la dynamique des situations.

La sémantique des combinateurs de la langue naturelle peut ainsi être finement analysée : résolution des anaphores, traitement de la causalité implicite aux pré-suppositions, etc. La technique des continuations, utilisant le calcul fonctionnel pour définir le sens des opérations des langages de programmation, est en train de trouver un nouveau champ d'application avec le traitement du sens de la langue naturelle.

Linguistique formelle et Calcul fonctionnel

L'interface syntaxe-sémantique est ainsi couvert de manière élégante par deux variétés de λ-calcul. Au sein du projet Calligramme de Nancy, Philippe de Groote développe ainsi une théorie des types permettant à la fois de vérifier la grammaticalité et de manipuler le sens de la langue, appelée "Abstract Categorial Grammars".

Aarne Ranta, à l'Université Chalmers, développe un système multilingue inspiré de la théorie des types sous le nom de "Grammatical Framework".

De très nombreux efforts de par le Monde utilisent des variantes du calcul fonctionnel pour la compréhension de la langue naturelle.

Conclusion

Le calcul fonctionnel (et ses dérivés) est une notation mathématique essentielle et donne des fondements uniformes à l'Informatique, à la Logique, et à la Linguistique.

Au delà du calcul fonctionnel, la logique linéaire offre l'attrait d'un système plus symétrique, où les liens axiomes offrent une possibilité d'interaction supplémentaire, pour remplacer l'arbre des calculs par un graphe où interragissent des processus communicants.

Merci de votre attention

Les notes de cours : "**Constructive Computation Theory** Course notes on λ-calculus" sont disponibles à l'URL `http://yquem.inria.fr/~huet/PUBLIC/CCT.pdf`. Ce cours est exécutable en Ocaml, et le source des programmes peut être téléchargé à l'URL `http://yquem.inria.fr/~huet/PUBLIC/LAMBDA.tar.gz`.

Le texte de ce séminaire est disponible à `http://yquem.inria.fr/~huet/PUBLIC/CdF09.pdf`.