La théorie de la fonctionnalité à la croisée des chemins entre Informatique, Logique et Linguistique

Gérard Huet

INRIA

Colloquium Sophia-Antipolis, 5 février 2004

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

- Calculabilité
- Turing
- Church
- $\lambda x \cdot (sin((mult\ deux)\ x))$

Une algèbre libre à 3 constructeurs très simple :

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

λ-calcul

- variable liée - un non-problème (de Bruijn)
- règle β - substitution
- $(\lambda x \cdot E\ X) \rhd E[x \leftarrow X]$
- congruence
- non déterminisme
- confluence
- unicité des formes normales

Complétude combinatoire

- $\lambda x \cdot (x\ x)$
- $(\lambda x \cdot (x\ x)\ \lambda x \cdot (x\ x))$
- $\{x \mid \neg x \in x\}$ Paradoxe!
- fonctions *partielles* récursives
- stratification par des types
- théorie des types, fondements
- Church's simple theory of types - more later

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)
```

Quicksort 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 λ-term [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 ?

Le λ-calcul comme langage de programmation

Le λ-calculus est un langage de programmation de très haut niveau.

Le λ-calculus est le langage orienté objet ultime.

Le λ-calculus est un langage de programmation de très bas niveau.

Le λ-calculus peut exprimer la sémantique d'un langage de programmation arbitraire.

Le λ-calculus est un bon support de cours à la théorie des fonctions récursives - dans un cadre constructif non extensionnel. Voir mon cours de DEA `http://pauillac.inria.fr/~huet/CCT/`

Impact sur les vrais langages de programmation

Peter Landin, qui connaissait bien le λ-calcul, a eu une influence déterminante au sein du comité Algol 60.

John McCarthy s'est inspiré du λ-calcul dans la conception du langage LISP - mais la liaison dynamique de LISP était contraire à l'esprit du λ-calcul où les variables sont liées statiquement. Scheme, le successeur de LISP, s'en rapproche davantage.

Les vrais langages de programmation possèdent des types de base munis d'opérations correspondant aux processeurs physiques des ordinateurs, comme les entiers ou les flottants, ainsi que des pointeurs utilisant directement l'adressage de la mémoire. Le λ-calcul pur n'est pas approprié à cette transparence de l'architecture matérielle, mais il peut simuler des mécanismes impératifs arbitraires - c'est l'essence de la sémantique dénotationnelle.

Impact sur les vrais langages - suite

La famille ISWIM-ML-Haskell est elle directement inspirée du λ-calcul. Mais l'opérateur de récursion n'est pas codé en λ-calcul, etc.

Ceci est à contraster avec la famille CPL-BCPL-C, qui privilégie les calculs de pointeurs.

Les mécanismes de modules paramétriques sont plus ou moins des λ-calculs.

On stratifie avec des types

On évite les expressions paradoxales comme $(x\ x)$ en stratifiant les expressions avec des types explicitant la fonctionnalité des objets. Par exemple, la théorie des types simples de Church utilise des types formés à partir de types atomiques (int, bool) et du foncteur $\rightarrow$.

Par exemple, si on a le contexte de déclarations :
$[n : i][s : i \rightarrow i][plus : i \rightarrow (i \rightarrow i)]$ alors le terme $\lambda x \cdot (plus\ x\ (s\ n))$ est bien typé de type $i \rightarrow i$. Le système de typage utilise les règles d'inférence :

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

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

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

On stratifie avec des types – suite

Normalisation forte.

- Typage à la Church vs typage à la Curry
- Type principal
- Polymorphisme du let - ML (Milner)
- λ-calcul polymorphe - F (Girard)
- $\mathrm{NAT} = \forall A \cdot (A \to A) \to (A \to A)$

Les entiers sont des itérateurs. Leur type se lit directement depuis la présentation d'une algèbre de Peano par Curryfication :

$$Peano = \{N; S : N \to N; Z : N\}$$

NAT décrit l'algèbre de Peano initiale.

Fondements logistiques des mathématiques

On peut maintenant utiliser le λ-calcul typé comme calcul des prédicats d'ordre supérieur, c'est-à-dire avec des fonctions et fonctionnelles arbitraires. C'est le programme de la théorie des types de Church, qui simplifie l'approche de Russell dans Principia Mathematica.

$$\forall x \cdot P(x) \quad \equiv \quad \Pi(\lambda x \cdot (P\ x))$$

En arithmétique, le principe de récurrence devient un axiome (et non un schéma d'axiome comme en arithmétique de premier ordre).

On peut même *définir* l'égalité selon Leibniz:

$$= \quad \equiv \quad \lambda x \cdot \lambda y \cdot \forall P \cdot (P\ x) \Rightarrow (P\ y)$$

Vers la mécanisation des mathématiques

Cette notation mathématique est *bien fondée* car on peut expliciter toutes les notions en les réduisant en leur forme normale.

On peut même dans une certaine mesure automatiser les raisonnements mathématiques dans cette logique – Constrained Resolution (1972).

Un autre point de vue : $r \rhd R$

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

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

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

Les types sont devenus des propositions logiques, les variables nomment des axiomes, et les λ-termes sont maintenant des *preuves*. C'est la présentation du fragment implicationnel du calcul propositionnel sous forme de *déduction naturelle*.

C'est tellement simple qu'il a fallu 40 ans pour s'apercevoir de cet isomorphisme. Howard, génial découvreur de cette correspondance, qui s'étend directement à toute la logique constructive, a été superbement ignoré.

On mélange tout

On obtient le Calcul des Constructions en ajoutant le calcul des propositions et le calcul des preuves, pour obtenir un langage uniforme de description des mathématiques (Coquand, 1985).

En ajoutant des types inductifs pour décrire les données finitaires, on obtient le Calcul des Constructions Inductives (Mohring, 1990). En ajoutant des types co-inductifs, on peut modéliser les processus communicants (Gimenez, 1995). C'est le programme Coq.

Application industrielle typique : certification de l'environnement d'exécution SUN de JavaCard (Trusted Logic, 2003).

Le λ-calcul sert à relever les défis stratégiques

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 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.

L'interface syntaxe-sémantique est ainsi couvert de manière élégante par deux variétés de λ-calcul. C'est le programme de travail des projets Calligramme et Signes.

Conclusion

Le λ-calcul (et ses dérivés) donne des fondements uniformes à l'Informatique, à la Logique, et à la Linguistique.

Un peu de préhistoire

Autrefois, les ancêtres :

- Shönfinkel 1930
- Gentzen 1935
- Church 1940
- Curry 1950

Dans les années 60, les précurseurs :

- Böhm 67 arbres de Böhm (séparabilité)
- de Bruijn 68 indices de de Bruijn (syntaxe abstraite)
- Landin 67 The next 700 programming languages

Dans les années 70, les visionnaires incompris :

- Girard 71 Système F (polymorphisme)
- Reynolds 74 λ-calcul polymorphe
- de Bruijn 75 Automath
- Howard (Correspondance)

Les sémanticiens :

- Scott 1971 $D\infty$
- Plotkin 1975 T^{ω}
- Wadsworth 1972 (continuité)
- Lévy 1978 (optimalité)
- Barendregt 1978 (synthèse)
- Milner (LCF, ML)
- Kahn, Berry, Curien (séquentialité)

Dans les années 80, la théorie des types :

- Martin-Löf (théorie constructive des types)
- Constable (Extraction de programmes à partir de preuves)
- Huet 1984 (Constructive Engine)
- Coquand 1985 (Calcul des Constructions)
- Paulin 1992 (Coq)

Remarque. En 1984, Gilles Kahn organisait la Conférence sur les Types à Sophia-Antipolis.