The functional calculus and its applications to algorithms, proofs, and linguistics

Gérard Huet

INRIA

IIT Delhi, February 21st 2011

Mathematical notation

- the function in x with value $sin(2x)$
- $x \mapsto sin(2x)$
- $\int sin(2x)dx \quad \star \int(x \mapsto sin(2x))$
- $\star(x \mapsto sin(2x)) \circ (x \mapsto x^2)$
- $\star x \mapsto (y \mapsto sin(x+y))$
- $\forall \quad \partial \quad \sum$
- $\{x \mid P(x)\}$
- Bourbaki's "assemblages"

λ-notation

We take seriously mathematical notation, with precise management of environnements in which we specify axioms, hypotheses, definitions, etc. The central notation is the λ-notation due to Alonzo Church (1936).

This notation, and the λ-calculus based on it, were first used for the definition of computable functions.

It is a very simple algebra with 3 constructors:

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

In order to explain how λ is a *binding operator*, it is essential to abstract variable names from the notation.

Functional terms

We abstract variables by replacing them with a relative notation within fonctional terms, which are oriented trees with two sorts of branching, of respective arity 1 and 2. The leaves of the trees are natural numbers in $\mathbb{N} = \{0, 1, 2, ...\}$.

We call *abstractions* unary branchings and *applications* the binary ones. Integres are *relative variables* (aka de Bruijn indices). Each variable is relative to its binding abstraction, since it denotes its depth in the abstraction layers.

Functional terms are built in an environnement of depth n, as follows.

Functional terms (2)

$$n+1 \vdash n$$

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

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

Closed terms (*combinators* are built at depth 0. For instance, $I = \Lambda 0$, and thus $(I\ I) = (\Lambda 0\ \Lambda 0)$, $\Delta = \Lambda(0\ 0)$, $\Omega = (\Delta\ \Delta)$, etc. In a closed term, all variables denote. *Open terms* have free variables. Terms (open and closed) constitute the *free functional algebra* generated by $\mathbb{N}$.

Perlis' law

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

In other words, the free variables of a term are the variables bound it its defining context. Without this precaution, we would get an ambiguous notation.

Successor

We define term $Succ^k\ E$, for E a term and $k \geq 0$, as :
$Succ^k\ E = Succ_0^k\ E$, with $Succ_n^k$ defined as:

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

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

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

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

Substitution

Let T be a term, we define term $Subst\ T\ E = Subst_0^T\ E$ as:

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

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

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

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

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

The Substitution theorem

Theorem.

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

or, more precisely:

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

This theorem is the central tool in the confluence of the functional calculus.

Details are available in:
Constructive Computation Theory. Course notes on λ-calculus. See `http://yquem.inria.fr/~huet/PUBLIC/CCT.pdf`. This course is executable in Ocaml. The program sources may be downloaded at URL: `http://yquem.inria.fr/~huet/PUBLIC/LAMBDA.tar.gz`.

Functional calculus

We associate to the functional algebra a computation relation called β-reduction. It is defined as the congruence generated by:

$$(\Lambda M\ N) \rightarrow_\beta \mathit{Subst}\ N\ M$$

Theorem (Church and Rosser). β-reduction is confluent.

Corollary. The normal form of a term (irreducible form obtained by iterating reduction) is unique, when it exists.

Corollary. The functional calculus, although non deterministic, is *determinate*.

Functional calculus applied to mathematical notation

We may now get uniform mathematical notations as functional expressions:

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

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

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

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

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

Combinatorial completeness

- $\lambda x \cdot (x\ x)$
- $(\lambda x \cdot (x\ x)\ \lambda x \cdot (x\ x))$
- $\{x \mid \neg x \in x\}$ Paradox!
- *partial* recursive functions and functionals
- stratification by types
- type theory, foundations
- Church's simple theory of types - more later

Programming foundations

NB. We use ML as meta-notation.

Notation : [x]M $= \lambda x \cdot M$, [x,y]M $= \lambda x \cdot \lambda y \cdot M$, (M N P) = ((M N) P)

```
(* 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)))>>;
```

Church's arithmetics

```
(* 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 in λ-calculus

```
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 [assembly language]

```
_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 [machine langage]

Its bare term, without syntactic sugar:

```
_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: What is the machine?

λ-calculus a a programming langage

λ-calculus is a very high level programming langage.

λ-calculus is the ultimate object-oriented language.

λ-calculus is a very low level programming langage.

λ-calculus may be used to define the formal semantics of an arbitrary programming language.

λ-calculus is a good support to teach the theory of recursive functions - within a non-extensional rigorous constructive framework - superior to classical set theory. Please visit `http://yquem.inria.fr/~huet/CCT/`

Impact on real programming languages

Peter Landin, who knew well λ-calculus, had a significant influence within the Algol 60 commitee that designed the language 50 years ago.

John McCarthy was inspired by λ-calculus in the design of LISP - although LISP's dynamic binding de LISP was not consistent with λ-calculus where variables are statically bound. Scheme, LISP's successor, is more in its spirit.

Real programming languages possess datatypes whose operations correspond to the physical processors of digital computers, such as integers and floating point numbers, as well as pointers using directly the memory addressing mechanisms. λ-calcul is not appropriate to this transparency of the material architecture of computers, but it may simulate arbirarily complex imperative mechanisms - this is the essence of denotational semantics.

Impact on real languages - more

The ISWIM-ML-Haskell family is directly inspired from λ-calculus. But the recursion operation is not encoded in λ-calculus, but a primitive operation.

This is in contrast with the CPL-BCPL-C family, that emphasizes pointer computation for fast vector addressing.

More recently, parametric module mechanisms have been inspired more or less directly by λ-calculus.

Let us stratify with types

We avoid paradoxical combinators such as $(x\ x)$ by the stratification of terms with types that express the functionality of their denotation. Thus, Church's simple theory of types uses types constructed from atom types (int, bool) with the $\rightarrow$ functor.

For instance, assuming the declaration context:
$[n : i][s : i \rightarrow i][plus : i \rightarrow (i \rightarrow i)]$, the term $\lambda x \cdot (plus\ x\ (s\ n))$ is well-formed with type $i \rightarrow i$. This typing system uses the inference rules:

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

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

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

More on type stratification

Strong normalisation.

- Typing Church's way versus typing Curry's way
- Principal typing (Hindley)
- Let polymorphism - ML (Milner)
- polymorphic λ-calculus - F (Girard)
- NAT $= \forall A \cdot (A \to A) \to (A \to A)$

Natural numbers are seen as iterators. Their type may be directly read as Curryfication of a Peano-algebra presentation:

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

NAT is thus directly seen as the initial Peano algebra. This was generalized to algebraic datatypes by Böhm and Berarducci.

Logical foundation of mathematics

We may now use typed λ-calculus as a higher-order predicate calculus, manipulating arbitrary functions and functionals. This is the program set by Church in his Simple Type Theory, simplifying the more complex approach to foundation of Russell and Whitehead in Principia Mathematica.

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

In arithmetics, the induction principle is now directly expressed as an axiom (as opposed to an infinitary axiom schema like in first order arithmetics).

We may even *define* equality along with Leibniz:

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

Towards a mechanization of mathematics

This treatment of mathematics is *well founded* since we may make explicit all the notions by reducing them to their normal form (although this may be arbitrarily costly).

We may even to a certain extent give an automatic treatment to mathematical deduction in this logic – see Constrained Resolution (1972).

Another viewpoint: $r \rhd R$

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

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

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

Types are now seen as logical propositions, variables name axioms, and λ-terms denote now *proofs*. We get the standard presentation of the implicational fragment of propositional calculus in Gentzen's *natural deduction*.

This is so simple that it took 40 years to notice this isomorphism. Howard, who showed the extension of this correspondance to predicate calculus, which extends to higher-order logic, was just ignored. Similarly with Nicolaas de Bruijn.

We mix all those ideas

We obtain the Calculus of Constructions by superposing the calculus of propositions and the calculus of proofs, obtaining thus a uniform language for describing mathematics (Coquand, 1985).

Adding inductive types to describe datastructures, we get the Calculus of Inductive Constructions (Mohring, 1990). Adding co-inductive types, we may describe communicating systems (Gimenez, 1995). This programme is being developed with the Coq proof environment.

A typical industrial application: certification of the SUN JavaCard run-time environnment (Trusted Logic, 2003). A typical application to formal mathematics is the proof by Gonthier of the 4-color theorem (2004).

We dualize the arrow of functionality

$A\backslash B$ types functions taking their argument $(b : B)$ to their left
A/B types functions taking their argument $(b : B)$ to their right.

$$John\ loves\ Mary.$$

$$loves : (NP\backslash S)/NP$$
$$John : NP$$
$$Mary : NP$$

$$loves\ Mary : NP\backslash S$$
$$John\ loves\ Mary : S$$

The terms of this symetrized λ-calculus may be used as phrase structure syntax operators, and types are syntactic categories. Such λ-terms are constrained with a linearity condition. We thus get categorial grammars (Lambek 1954).

After syntax, semantics of natural language

Montague's grammars express a notion of semantics of natural languages, where the logical contents of a sentence is expressed in Church's type theory.

The syntax-semantics interface is thus covered in an elegant fashion by two varieties of λ-calculus. This leads to important recent developments in computational linguistics research.

Conclusion

λ-calculus and its derivatives provide uniform foundations to Informatics, Logic and Linguistics.

Prehistory

Once upon a time, the early logicians:

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

In the 60's, the precursors:

- Böhm 67 Böhm trees (separability)
- de Bruijn 68 de Bruijn's indices (abstract syntax)
- Landin 67 The next 700 programming languages

In the 70's, the ignored visionaries:

- Girard 71 System F (polymorphism)
- Reynolds 74 polymorphic λ-calculus
- de Bruijn 75 Automath
- Howard (Correspondance)

Semanticists:

- Scott 1971 $D\infty$
- Plotkin 1975 T^ω
- Wadsworth 1972 (continuity)
- Lévy 1978 (optimality)
- Barendregt 1978 (synthesis)
- Milner (LCF, ML)
- Kahn, Berry, Curien (sequentiality)

In the 80's, type theory:

- Martin-Löf (constructive type theory)
- Constable (Extracting programs from proofs, νPRL)
- Huet 1984 (Constructive Engine)
- Coquand 1985 (Calculus of Constructions)
- Paulin 1992 (Coq)
- Coq V8.3 2010 (`http://coq.inria.fr/`)