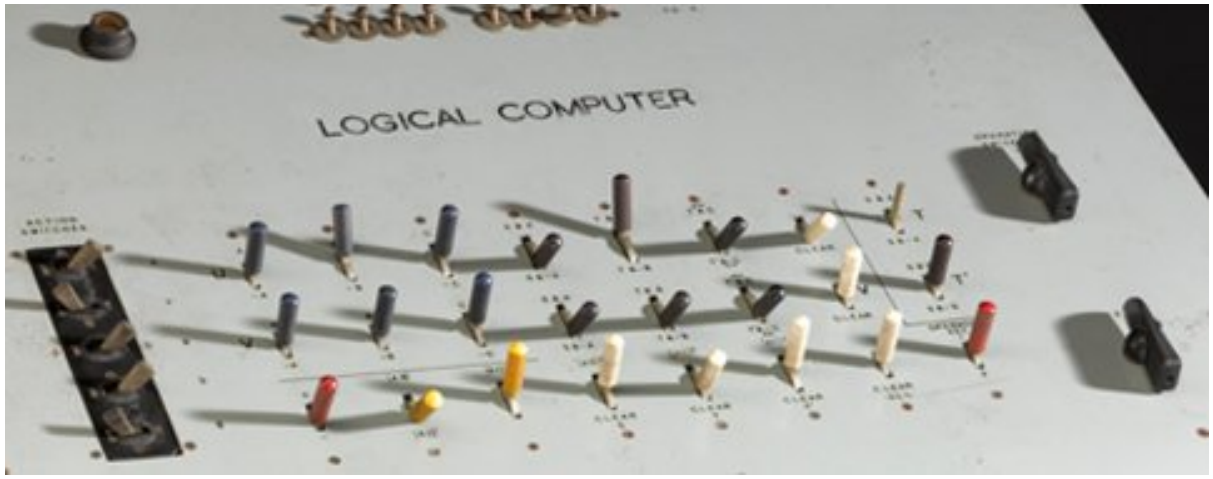
INF551

Computational Logic:

Artificial Intelligence in Mathematical Reasoning



Stéphane Graham-Lengrand

Stephane.Lengrand@Polytechnique.edu

Lecture VIII

The λ-calculus: arithmetic and computability

One preliminary remark about HOL

Take it for granted:

- **Type-checking**:

  The problem of knowing, for a λ-term t, a context Δ and a type A, whether $\Delta \Vdash t : A$, is decidable

- **Typability**:

  The problem of knowing, for a λ-term t, whether it is typable (there exist Δ and A such that $\Delta \Vdash t : A$), is also decidable

Remark: The typing system is a fragment of Caml's.

If the Caml compiler can type-check, so can we.

Corollary: Proof checking in HOL is decidable

In particular, it can be decided whether an application of rule

$$\frac{\Gamma \vdash_\Delta Q \quad \Delta \Vdash P : \mathsf{Prop}}{\Gamma \vdash_\Delta P} \; P \longleftrightarrow^* Q$$

is correct, because of confluence and strong normalisation of P and Q

Contents

I. Arithmetic in Higher-Order Logic

Representation of natural numbers

Representation is guided by the desire to define functions by induction

3 is iterating a function three times

$$\underline{3} = \lambda x.\lambda f.(f\ (f\ (f\ x)))$$

$$\underline{p} = \lambda x.\lambda f.(\underbrace{f\ (f\ \ldots\ (f}_{p \text{ times}}\ x)\ldots))$$

If h is the function that doubles its argument, what is $(\underline{n}\ \underline{1}\ h)$?

Every $\underline{p}$ can be typed:

$$\Vdash \underline{p} : a \to (a \to a) \to a$$

Let us abbreviate $a \to (a \to a) \to a$ by a'.

Every $\underline{p}$ is irreducible

Easy to check: every closed irreducible λ-term of type a' is of the form $\underline{p}$

So far we have seen four representations of natural numbers

3 is $S(S(S(0)))$ (Peano)

3 is what all the sets of three elements have in common (Cantor)

3 is $\{0, 1, 2\}$ (Von Neumann)

3 is $\lambda x.\lambda f.(f\ (f\ (f\ x)))$ (Church)

Representation of functions, successor, addition

Definition

F (λ-term) represents a total function f from $\mathbb{N}^n$ to $\mathbb{N}$

if for all $p_1, \ldots, p_n, q$ such that $f(p_1, \ldots, p_n) = q$, we have $(F\ \underline{p_1}\ \ldots\ \underline{p_n}) \longrightarrow^* \underline{q}$

Successor: $\quad \mathsf{SUC} := \quad \lambda n.\lambda x.\lambda f.f\ (n\ x\ f)$

Addition: $\quad \mathsf{PLUS} := \quad \lambda p.\lambda q.\lambda x.\lambda f.p\ (q\ x\ f)\ f$

Multiplication: $\quad \mathsf{TIMES} := \quad \lambda p.\lambda q.\lambda x.\lambda f.p\ x\ (\lambda y.q\ y\ f)$

Clearly,

$\Vdash \mathsf{SUC} : a' \to a'$

$\Vdash \mathsf{PLUS} : a' \to a' \to a'$

$\Vdash \mathsf{TIMES} : a' \to a' \to a'$

Peano's axioms

The following theorem can be proved in HOL:

$$\vdash (\text{TIMES}\ \underline{45}\ \underline{67}) = \underline{3015}$$

in just a couple of steps!

. . . without axioms

. . . just using computing power of λ-term reduction

in that sense, HOL much more computer-friendly than set theory

The following Peano's axioms can be proved in HOL (see practical):

$$\vdash \forall_{a'} n\ (\text{PLUS}\ \underline{0}\ n) = n$$

$$\vdash \forall_{a'} n \forall_{a'} m\ (\text{PLUS}\ (\text{SUC}\ n)\ m) = \text{SUC}\ (\text{PLUS}\ n\ m)$$

$$\vdash \forall_{a'} n \forall_{a'} m\ (\text{TIMES}\ \underline{0}\ m) = \underline{0}$$

$$\vdash \forall_{a'} n \forall_{a'} m\ (\text{TIMES}\ (\text{SUC}\ n)\ m) = \text{PLUS}\ m\ (\text{TIMES}\ n\ m)$$

Just like in $\mathcal{ZF}^*$

But this time without axioms!

. . . well, almost

The first one: $\vdash \forall_{a'} n\ (\text{PLUS}\ \underline{0}\ n) = n$

actually requires a weak form of extensionality: $\forall_{a \to b} f\ (f = \lambda x.f\ x)$

Cannot be proved in HOL as defined last week. . . axiom?

But can be proved in HOL if the notion of reduction is extended

(2 reduction rules instead of one):

$$(\lambda x.t)\ u \longrightarrow_{\text{root}} \{{}^{u}\!/_{x}\}t$$

$$\lambda x.t\ x \longrightarrow_{\text{root}} t$$

All the theorems you know about λ-calculus still hold with the extra rule

(confluence, strong normalisation of simply-typed terms, subject reduction, etc)

Coq implements (an extension of) HOL.

version 8.3 integrates only the first rule

$\Longrightarrow$ weak extensionality needs to be added as an axiom

version 8.4 integrates both rules

$\Longrightarrow$ weak extensionality can be proved

Induction

For all integer p we have in HOL a proof of

$$\vdash \forall_{a' \to \mathsf{Prop}} P\ (P\ \underline{0} \Rightarrow (\forall_{a'} m, P\ m \Rightarrow P\ (\mathsf{SUC}\ m)) \Rightarrow P\ \underline{p})$$

But it can be shown (not in INF551) that in HOL there is no proof of

$$\vdash \forall_{a'} n \forall_{a' \to \mathsf{Prop}} P\ (P\ \underline{0} \Rightarrow (\forall_{a'} m, P\ m \Rightarrow P\ (\mathsf{SUC}\ m)) \Rightarrow P\ n)$$

i.e., you cannot prove (within HOL) that every inhabitant of a' satisfies every "hereditary property" (a predicate P such that $P\ \underline{0}$ and $\forall_{a'} m, P\ m \Rightarrow P\ (\mathsf{SUC}\ m)$)

Church's trick: Let's say that natural numbers are not all the inhabitants of a'

but only those satisfying every hereditary property, i.e., satisfying the predicate NAT_a:

$$\lambda n. \forall_{a' \to \mathsf{Prop}} P\ (P\ \underline{0} \Rightarrow (\forall_{a'} m, P\ m \Rightarrow P\ (\mathsf{SUC}\ m)) \Rightarrow P\ n)$$

Safety check: you can prove $\vdash\ \mathsf{NAT}_a\ \underline{0}$ and $\vdash\ \forall_{a'} n, (\mathsf{NAT}_a\ n) \Rightarrow (\mathsf{NAT}_a\ (\mathsf{SUC}\ n))$

Now when we want to quantify over natural numbers, we write

$$\forall_{a'} n, (\mathsf{NAT}_a\ n) \Rightarrow \cdots \qquad \text{or} \qquad \exists_{a'} n, (\mathsf{NAT}_a\ n) \wedge \cdots$$

And now we can prove the induction principle (see practical):

$$\vdash \forall_{a'} n, (\mathsf{NAT}_a\ n) \Rightarrow \forall_{a' \to \mathsf{Prop}} P\ (P\ \underline{0} \Rightarrow (\forall_{a'} m, P\ m \Rightarrow P\ (\mathsf{SUC}\ m)) \Rightarrow P\ n)$$

Missing Peano axioms

Two of peano's axioms are still missing

- Zero is the successor of noone: $\forall_{a'} n\ (\mathsf{NAT}_a\ n) \Rightarrow \neg(0 = \mathsf{suc}\ n)$

  cannot be proved (with or without the assumption $(\mathsf{NAT}_a\ n)$)

  But this can actually be proved (see practical) when a has two distinct elements:

  $$\vdash (\exists_a x \exists_a y\ \neg(x = y)) \Rightarrow \forall_{a'} n\ \neg(0 = \mathsf{suc}\ n)$$

  This is the case of Prop!

  Therefore: $\vdash \forall_{\mathsf{Prop}'} n\ \neg(0 = \mathsf{suc}\ n)$

- Successor is injective:

  $$\forall_{a'} n\ (\mathsf{NAT}_a\ n) \Rightarrow \forall_{a'} m\ (\mathsf{NAT}_a\ m) \Rightarrow (\mathsf{suc}\ n = \mathsf{suc}\ m) \Rightarrow n = m$$

  No trick there, we have to add it as an axiom

Church, Turing and Goedel crash the party again!

Let HOL+ be HOL with

- the injectivity of successor
- Weak Extensionality (either as an axiom or as part of the reduction relation)

We have seen that HOL+ can fully express Peano's arithmetic

(i.e., can prove the theory $\mathcal{T}_0$)

Sanction: If HOL+ is consistent (i.e., has a model, which would be an $\mathbb{N}$-model), then

- Provability in HOL+ is undecidable (Church's theorem)
- There is a Goedel formula in HOL+ (or in HOL) (Godel's theorem)

  (cannot be proved and neither can be its negation)

The second point is due to the fact that proof-checking in HOL+ is still decidable

Consistency

Well. . . is HOL+ (or even HOL) consistent? Nobody has found a contradiction yet

Simple types prevent the construction of paradoxes such as Russel's:
the term $(\lambda x.x\ x)\ (\lambda x.x\ x)$ with infinite reduction is banned
since $x\ x$ cannot be typed!

Can we prove that HOL+ (or even HOL) is consistent?

Well, it depends which meta-logic we work in:

Working in set theory, we **have proved** that Peano's Arithmetic is consistent
(we have constructed an $\mathbb{N}$-structure in $\mathcal{ZF}^*$)

Still working in set theory, we **could prove** that HOL+ (and HOL) is consistent
(would only take a couple of hours)
. . . and it heavily relies on the strong normalisation of λ-terms that are used

Morality: deep connection between consistency and strong normalisation

II. The λ-calculus beyond simple types

Turing-completeness?

You will have noticed that our λ-terms are particular (and simple) Caml programs

Caml, like every decent programming language, is Turing-complete

Question: is the simply-typed λ-calculus Turing-complete?

Remember the Generalised Halting Problem (Lecture 3):

Let T be a decidable subset of programs such that

every program in T always terminates.

Then T is not Turing-complete

(there is a computable function not represented by a program in T)

Fact 1: Being a λ-term of type $a' \to a'$ is a decidable property (Typing is decidable)

Fact 2: If $\Vdash t : a' \to a'$, then for every numeral $\underline{p}$ (of type a'), we have $\Vdash t\,\underline{p} : a'$

. . . and therefore $t\,\underline{p}$ is strongly normalising ($\Longrightarrow$ terminates)

Conclusion: The simply-typed λ-calculus is not Turing-complete

What does the simply-typed λ-calculus lack for Turing-completeness?

The source of non-termination in your usual programming language.

For instance:

- the `while` loop in imperative languages (e.g., C)
- the general recursive calls `let rec ...= ...in ...`

  in functional languages (e.g., Caml)

In summary:

strong normalisation deeply connected to the consistency of the logic using λ-terms

strong normalisation incompatible with Turing-completeness

Very difficult (impossible?) to have a logic integrating all computations from a Turing-complete language

Let us investigate how to make the λ-calculus Turing-complete

independently from any logic

Representation of functions in general

Our definition of "representing a function" was designed for total functions

Let us extend this definition for partial functions as well

Definition

F (λ-term) represents a function f from $\mathbb{N}^n$ to $\mathbb{N}$

- if for all $p_1, \ldots, p_n, q$ such that $f(p_1, \ldots, p_n) = q$,

$$(F\ \underline{p_1}\ \ldots\ \underline{p_n}) \longrightarrow^* \underline{q}$$

- and if for all $p_1, \ldots, p_n$ such that f is not defined on $p_1, \ldots, p_n$,

$(F\ \underline{p_1}\ \ldots\ \underline{p_n})$ does not reduce to an irreducible term

This defines a notion of "computable functions" (those represented by some λ-terms)

3 equivalent notions

- Recursive functions - Gödel, 1933
- λ-calculus - Church, 1936
- Turing machines, 1937

Rosser (1939): those three computational models coincide

(they identify as “computable” the same set of functions)

In the next section, we prove that recursive functions can be represented by λ-terms

III. Representing recursive functions in the λ-calculus

Projections, zero, successor, composition

We have already done most of the work!

Projections: $\lambda x_1. \ldots. \lambda x_n.x_i$

Constant functions returning 0: $\lambda x_1. \ldots. \lambda x_n.\underline{0}$

Successor $\lambda n.\lambda x.\lambda f.f\ (n\ x\ f)$

Composition:

If F represents $f : \mathbb{N}^m \longrightarrow \mathbb{N}$ and each G_i represents $g_i : \mathbb{N}^n \longrightarrow \mathbb{N}$, then

$$\begin{array}{rccc} h: & \mathbb{N}^n & \longrightarrow & \mathbb{N} \\ & (p_1, \ldots, p_n) & \mapsto & f(g_1(p_1, \ldots, p_n), \ldots, g_m(p_1, \ldots, p_n)) \end{array}$$

is represented by

$$H = \lambda x_1. \ldots \lambda x_n.(F\ (G_1\ x_1 \ldots x_n)) \ldots (G_m\ x_1 \ldots x_n))$$

This is a bit too naive, though

If g not defined on 4 and h is the constant function returning 0,

then the composition $f = h \circ g$ is not defined on 4

but

$$(F\ \underline{4}) = (\lambda y.((\lambda x.\underline{0})\ (G\ y))\ \underline{4}) \longrightarrow^* ((\lambda x.\underline{0})\ (G\ \underline{4})) \longrightarrow^* \underline{0}$$

The trick: $\&$

$t\&u$ is a λ-term meaning "t, provided that u reduces to some $\underline{p}$"

We can define $t\&u := (u\ t\ (\lambda x.x))$

It is easy to check that

- $t\&\underline{p} \longrightarrow^* t$
- $t\&u$ does not reduce to an irreducible term if u does not

Representing recursive functions - part 1

Projections: $\lambda x_1. \ldots. \lambda x_n.((((x_i \& x_1) \& \ldots \& x_{i-1}) \& x_{i+1}) \& \ldots \& x_n)$

Constant functions returning 0: $\lambda x_1. \ldots. \lambda x_n.((\underline{0} \& x_1) \& \ldots \& x_n)$

Successor: $\lambda n.\lambda x.\lambda f.(f\ (n\ x\ f))$

Plus: $\lambda p.\lambda q.(((\lambda x.\lambda f.(p\ (q\ x\ f)\ f)) \& p) \& q)$

Times: $\lambda p.\lambda q.(((\lambda x.\lambda f.(p\ x\ (\lambda y.(q\ y\ f)))) \& p) \& q)$

$\chi_<$: $\lambda p.\lambda q.(((p\ (K\ \underline{1})\ T\ (q\ (K\ \underline{0})\ T)) \& p) \& q)$

where $K = \lambda x.\lambda y.x$ and $T = \lambda g.\lambda h.(h\ g)$

Representing recursive functions - part 2

Composition:

$$\lambda x_1. \ldots \lambda x_n.(H\ (G_1\ x_1\ \ldots\ x_n))\ \ldots\ (G_m\ x_1\ \ldots\ x_n))$$

If then else:

$$\mathit{Ifz} = \lambda x.\lambda y.\lambda z.(x\ y\ \lambda z'.z)$$

Iterating a function represented by t forever:

$$Y_t = ((\lambda x.(t\ (x\ x)))\ (\lambda x.(t\ (x\ x))))$$

$$Y_t \longrightarrow\ (t\ Y_t) \longrightarrow\ (t\ (t\ Y_t)) \longrightarrow\ \cdots$$

Minimisation:

$$\lambda x_1. \ldots \lambda x_n.(Y_{G'}\ x_1\ \ldots\ x_n\ \underline{0})$$

where

$$G' :=\ \lambda f.\lambda x_1. \ldots \lambda x_n.\lambda x_{n+1}.(\mathit{Ifz}\ (G\ x_1\ \ldots\ x_n\ x_{n+1})\ x_{n+1}\ (f\ x_1\ \ldots\ x_n\ (S\ x_{n+1})))$$

Conclusion

Theorem:

If $f(p_1, ..., p_n) = q$, then $(F\ \underline{p_1}\ ...\ \underline{p_n}) \longrightarrow^*\ \underline{q}$

If f not defined on $p_1, ..., p_n$,

then $(F\ \underline{p_1}\ ...\ \underline{p_n})$ does not reduce to an irreducible term

Corollary: The untyped λ-calculus is Turing-complete

(since recursive functions are)

In the above constructions, which λ-terms can be typed?

Which one cannot?

Posterity of λ-calculus

Robin Milner: add to it

primitive numerals (4 operations + test), a primitive fixpoint and a `let` construct: PCF

Add to that

tree datatypes and pattern-matching: (Ca)ML, a real-life programming language

Coq uses a version of this where the fixpoint is controlled:

You can write (the equivalent of) `let rec f x = u`

when the type of `x` is a tree datatype,

but the only recursive calls of `f` allowed in `u` must be applied to arguments that are clearly subtrees of `x`

$\Longrightarrow$ ensures strong normalisation

$\Longrightarrow$ ensures consistency, but drops Turing-completeness

λ-calculus used as a basis to approach new domains: e.g., polymorphic λ-calculus, parallel λ-calculus, λ-calculus for quantum computations,...

IV. Intuitionistic logic (teaser)

Constructivism

Remember Kronecker's criticism of Cantor's work:

Cantor's inability to explicitly construct the sets he talked about

Example: the set of all sets

Fortunately, the existence of such a set leads to inconsistency (Russell)

and Zermelo-Fraenkel's set theory is a step towards constructivism:

Every set whose existence we claim needs to be justified by a construction, consisting of

- applying power set axiom or union axiom finitely many times

  (from the empty set or the set of natural numbers)
- and then using separation or replacement

Still, such constructions rely on axioms

Even better in HOL: there are no axioms!

Every function whose existence we claim is actually a λ-term, i.e., a computer program.

We can hardly get more explicit!

The drinker's theorem

Well. . . this is not completely true.

Let's come back to the drinker's theorem:

"There is always someone such that, if he drinks, everybody drinks"

Proof - Informal

Take the first guy you see, call it Bob.

Either Bob does not drink,

in which case he satisfies the predicate “if he drinks, everybody drinks”

. . . or Bob drinks, in which case we have to check that everybody else drinks

If this is the case, then again Bob is the person we are looking for

If we find someone who does not drink, call it Derek,

we change our mind and say that the guy we are looking for is Derek

Problem with this?

We have turned this into a formal proof of the formula $\exists x\ (\text{DRINKS}(x) \Rightarrow \forall y\ \text{DRINKS}(y))$

. . . using the rule

$$\frac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A}$$

We can develop the proof in predicate logic, in HOL, with or without a theory

Problem with this:

We have proved the theorem

. . . but we are still incapable of identifying the person satisfying the property

(or rather, our choice depends on the context)

In other words, we fail to provide a witness of existence

In other words, the logic we use does not have the witness property

The logic we use lacks a certain dose of constructivism

What's next?

Next Lecture:

- a way to modify every logical system we've talked about

  . . . to recover the witness property (and be more constructive)

- unveiling a new connection between proofs and programs:

  the Curry-Howard correspondence

Questions?