λ-calculus

G52LAC / COMP2012 - Spring 2019

Venanzio Capretta

Table of contents

History of λ-calculus

Hilbert's Program

David Hilbert (1928) asked the questions

- Is there a complete consistent formalization of all Mathematics?
- Is there an effective procedure to determine if a formula is provable? *(Entscheidungsproblem, the Decision Problem)*

Kurt Gödel (1931) answered the first question negatively

Incompleteness Theorems

- There is no complete axiomatization of Arithmetic
- Consistency of a formal system is not provable inside the system itself

Computability

Decision Problem

is there an *effective procedure* to decide if a formula is provable?

What is an effective procedure?

- An algorithm that takes a logical formula as input,
- performs a finite number of computation steps,
- returns *Yes* if the formula is provable, *No* otherwise.

There was no precise notion of algorithm or computation yet at the time

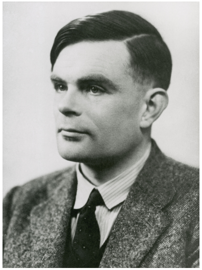
- There were no electronic computers
- *Computers* where human beings performing routine calculations (like the *Harvard computers* or the *NACA computer pool*)
- The notion of computability had to be defined in a purely mathematical way

The Harvard Computers

Origin of the λ-calculus

ALONZO CHURCH (1936) invents the λ-calculus

- A theory of pure functions
- The first complete model of computation
- A negative solution to the Decision Problem: *there is no algorithm to decide every mathematical formula*



ALAN TURING (1936, few months after Church) solves the same problem using machines

- Undecidability of the Halting Problem
- The Decision Problem is undecidable
- Equivalence of λ-calculus and Turing machines

Definition of λ-calculus

Definitions

In high-school we learn to define functions with a *mapping notation*.

A function on natural numbers can be defined as:

$$x \xmapsto{f} x^2 + 3$$

meaning: *f is the function that takes an input, which we call x, and returns as output its square plus three.*

Notation for functional abstraction: $f = \lambda x.\, x^2 + 3$.

Once defined, a function can be applied to any argument: $f(5)$ in high-school notation.

Notation for function application: $(f\ 5)$.

We compute the result by replacing the variable x with 5 and performing the operations:

$$(f\ 5) = ((\lambda x.\, x^2 + 3)\ 5) \rightsquigarrow 5^2 + 3 \rightsquigarrow^* 28$$

Formal Definition

In the language of λ-calculus we use only variables, abstraction and application.

No need for numbers, 3 and 5, or operations, squaring and addition.

Everything can be realized by pure functions.

Formal definition of λ-terms:

$$\begin{array}{rcll} \text{term} & ::= & x \mid y \mid z \mid \cdots & \text{variable names} \\ & \mid & (\lambda x.\, \text{term}) & \text{abstraction} \\ & \mid & (\text{term term}) & \text{application} \end{array}$$

(We omit unnecessary parentheses: *abstraction associates to the right, application associates to the left.*)

The only computation rule is β-reduction:

$$((\lambda x.\, t)\ u) \rightsquigarrow t[x := u]$$

(in term t, substitute every free occurrence of x with u)

Examples of λ-terms

- IDENTITY: $\mathsf{id} = \lambda x.\, x$

  The function that returns its input unchanged

$$\mathsf{id}\ a = (\lambda x.\, x)\ a \rightsquigarrow x[x := a] = a$$

- FIRST PROJECTION: $\mathsf{proj}_1 = \lambda x.\, \lambda y.\, x = (\lambda x.\, (\lambda y.\, x))$

  The function that, when applied to two arguments, returns the first

$$\begin{aligned}\mathsf{proj}_1\ a\ b \quad &= ((\mathsf{proj}_1\ a)\ b) = (((\lambda x.\, (\lambda y.\, x))\ a)\ b) \\ &\rightsquigarrow ((\lambda y.\, a)\ b) \rightsquigarrow a\end{aligned}$$

- SECOND PROJECTION: $\mathsf{proj}_2 = \lambda x.\, \lambda y.\, y$

  The function that returns the second argument

$$\mathsf{proj}_2\ a\ b \rightsquigarrow^* b$$

- COMPOSITION: $\mathsf{comp} = \lambda u.\, \lambda v.\, \lambda x.\, u\ (v\ x)$

  The function computes the composition of two functions

$$(\mathsf{comp}\ g\ f)\ a \rightsquigarrow^* g\ (f\ a)$$

Data Structures

Booleans

So far we only defined simple functions that perform projections and applications of their arguments

How can we represent mathematical objects and data structures?

IDEA:
To represent a set of objects, choose a separate λ-term for each object.
Make uniform choices that make it easy to define basic operations.

For the Booleans, we just need to pick two different terms.
Let's reuse the terms for first and second projection.
BOOLEANS:

$$\begin{aligned}\mathsf{true} &= \lambda x.\,\lambda y.\,x\\ \mathsf{false} &= \lambda x.\,\lambda y.\,y\end{aligned}$$

Underlying idea: a Boolean is a way of choosing between two arguments.
This makes it easy to define conditional choice $\mathsf{if} - \mathsf{then} - \mathsf{else} -$.

Conditionals and Connectives

Since the Booleans are just first and second projection,
the if – then – else – operation is just the application of the Boolean test to the two branches:

$$\text{if} = \lambda b.\, \lambda u.\, \lambda v.\, b\ u\ v$$

When the variable b is instantiated with the Boolean value, the term reduces to the correct branch:

$$\begin{aligned} \text{if true}\ t_1\ t_2 &= (\lambda b.\, \lambda u.\, \lambda v.\, b\ u\ v)\ \text{true}\ t_1\ t_2 \\ &\leadsto^* \text{true}\ t_1\ t_2 = (\lambda x.\, \lambda y.\, x)\ t_1\ t_2 \\ &\leadsto^* t_1 \end{aligned}$$

$$\text{if false}\ t_1\ t_2 \quad \leadsto^* t_2$$

The Boolean connectives can be realized either as applications of the conditional or directly:

$$\text{and} = \lambda a.\, \lambda b.\, \text{if}\ a\ b\ \text{false} \qquad \text{or}$$
$$\text{and} = \lambda a.\, \lambda b.\, a\ b\ \text{false}$$

Church Numerals

To represent the Natural Numbers,
we must choose infinitely many distinct terms

One way to do it is with Church Numerals:

$$\begin{aligned}
\overline{0} &= \lambda f.\, \lambda x.\, x \\
\overline{1} &= \lambda f.\, \lambda x.\, f\ x \\
\overline{2} &= \lambda f.\, \lambda x.\, f\ (f\ x) \\
\overline{3} &= \lambda f.\, \lambda x.\, f\ (f\ (f\ x)) \\
&\vdots \\
\overline{n} &= \lambda f.\, \lambda x.\, \underbrace{f\ (\cdots (f}_{n \text{ times}}\, x) \cdots)
\end{aligned}$$

The numeral $\overline{n}$ is an iterator:
It takes a function f and an argument x and iterates f n times on x.

This makes it easy to define functions by recursion.

Arithmetic Operations

The first simple operation is the successor: just add one extra iteration:

$$\text{succ} = \lambda n.\, \lambda f.\, \lambda x.\, f\ (n\ f\ x)$$

Other operations can be defined directly or iterating the previous one:

$$\text{plus} = \lambda n.\, \lambda m.\, \lambda f.\, \lambda x.\, n\ f\ (m\ f\ x)$$

or $\quad \text{plus} = \lambda n.\, n\ \text{succ}$

$$\text{mult} = \lambda n.\, \lambda m.\, \lambda f.\, n\ (m\ f)$$

or $\quad \text{mult} = \lambda n.\, \lambda m.\, n\ (\text{plus}\ m)\ \overline{0}$

$$\text{exp} = \lambda n.\, \lambda m.\, m\ n \qquad \text{(isn't this amazing?!)}$$

or $\quad \text{exp} = \lambda n.\, \lambda m.\, m\ (\text{mult}\ n)\ \overline{1}$

Exercise:

Try to define predecessor, (cut-off) subtraction, factorial.

Pairs

A pair is an operator that offers two arguments to any function:

$$\langle a, b \rangle = \lambda x.\, x\ a\ b$$

Projecting from a pair is done by application to a Boolean:

$$\mathsf{fst} = \lambda p.\, p\ \mathsf{true} \qquad \mathsf{snd} = \lambda p.\, p\ \mathsf{false}$$

These definition satisfy:

$$\mathsf{fst}\ \langle a, b \rangle \leadsto^* a \qquad \mathsf{snd}\ \langle a, b \rangle \leadsto^* b$$

But the surjective pairing property is not always true:

$$\langle \mathsf{fst}\ p, \mathsf{snd}\ p \rangle \not\leadsto^* p$$

(Exercise: find a term p for which $\langle \mathsf{fst}\ p, \mathsf{snd}\ p \rangle$ does not reduce to p)

There is no encoding of pairs that satisfies surjective pairing.

Tuples and Lists

Tuples can be defined similarly with several arguments
or by iterated pairing

$$\langle a, b, c\rangle = \lambda x.\, x\ a\ b\ c \qquad \text{or } \langle a, b, c\rangle = \langle a, \langle b, c\rangle\rangle$$
$$\langle a_0, \ldots, a_n\rangle = \lambda x.\, x\ a_0\ \cdots\ a_n \qquad \text{or } \langle a_0, \ldots, a_n\rangle = \langle a_0, \langle a_1, \ldots, a_n\rangle\rangle$$

Lists: we can also use iterated pairs, but we need the empty list.
One implementation is this:

$$\begin{aligned}\mathsf{nil} &= \mathsf{false}\\ \mathsf{cons}\ h\ t &= \langle h, t\rangle\end{aligned} \qquad [a_0, a_1, \ldots, a_n] = \langle a_0, \langle a_1, \cdots \langle a_n, \mathsf{false}\rangle\rangle\rangle$$

The basic operations on lists are:

$$\mathsf{head} = \mathsf{fst} \qquad \mathsf{tail} = \mathsf{snd} \qquad \mathsf{isNil} = \lambda l.\, l\ (\lambda h.\, \lambda t.\, \lambda x.\, \mathsf{false})\ \mathsf{true}$$

Exercise: Define a representation of lists as iterators, in the same style as Church Numerals.