Lambda calculus: Syntax

Jacques Garrigue, 2023/12/12

In 1920, Schönfinkel, a Russian logician, invented *combinatory logic*, which was to become *lambda-calculus* through the works of Curry and Church. As its original name shows, the goal was the formal manipulation of logical formulas. However, it became later connected to computer science, and provides a theoretical basis for *functional programming languages*, starting with Lisp in the 1950s. Despite its very simple definition it has a strong expressive power, and is often used as model for the theoretical study of programming languages.

1 Term Rewriting

The simplest definition of λ-calculus is as a term rewriting system. In term rewriting, we seen computation as the rewriting of part of terms through *rewriting rules*. For instance, here is a formalization of simple arithmetic.

Terms

$$\begin{array}{rcll} E & ::= & 0 & \text{zero} \\ & \mid & S(E) & \text{successor, i.e. } S(x) \text{ means } x+1 \\ & \mid & E+E & \text{addition} \\ & \mid & E-E & \text{substraction} \end{array}$$

Rewriting rules When x and y are arbitrary terms, the following rules define addition and substraction on natural numbers.

$$\begin{array}{lcl} x+0 & \rightarrow & x \\ x+S(y) & \rightarrow & S(x+y) \\ 0-x & \rightarrow & 0 \\ x-0 & \rightarrow & x \\ S(x)-S(y) & \rightarrow & x-y \end{array}$$

A term E is in *normal form* if no rule applies to it. The above rules are sometimes called *reduction rules.*

Example 1 (rewriting)

$$(S(0)+S(0))-S(0) \rightarrow S(S(0)+0)-S(0) \rightarrow S(S(0))-S(0) \rightarrow S(0)-0 \rightarrow S(0)$$

2 Syntax of lambda-calculus

Definition 1 *A* **λ-term** M *must be of the three following forms:*

$$\begin{array}{rcll} M & ::= & x & \text{variable} \\ & \mid & (\lambda x.M) & \text{abstraction} \\ & \mid & (M\ M) & \text{application} \end{array}$$

The variable x intuitively represents a value that should be bound in the environment. We will see how computation substitutes it with another λ-term.

$(\lambda x.M)$ binds the variable x if it appears in M. $f = \lambda x.M$ can be seen as a function, whose definition is $f(x) = M$. However, the λ notation avoids the need to give a name to this function. We will omit parentheses when $\lambda x.M$ is not part of an application.

$(M_1\ M_2)$ represents function application. This is similar to the usual notation $M_1(M_2)$, but M_1 need not be a variable, it can be any λ-term. We omit parentheses when this application is not the argument of another application, so that $\lambda x.M_1\ M_2$ means $(\lambda x.(M_1\ M_2))$ and $M_1\ M_2\ M_3$ means $((M_1\ M_2)\ M_3)$, i.e. application associates to the left.

Mixing the above grammar with arithmetic,

$$f(2) \quad \text{when} \quad f(x) = x + 1$$

can be written directly as

$$(\lambda x.x + 1)\ 2$$

Free variables and substitution In $\lambda x.M$, all occurences of x in M are said to be *bound*. If a variable x appears in a term M without being bound, it is sai to be *free*. The set of free variables of M is defined inductively as follows.

$$\begin{array}{lcl} FV(x) & = & \{x\} \\ FV(\lambda x.M) & = & FV(M) \setminus \{x\} \\ FV(M\ N) & = & FV(M) \cup FV(N) \end{array}$$

Substitution replaces such free variables with other λ-terms. $([N/x]M)$ replaces all free occurences of x in M with N.

$$\begin{array}{lcll} ([N/x]x) & = & N & \\ ([N/x]y) & = & y & x \neq y \\ ([N/x]\lambda x.M) & = & \lambda x.M & \\ ([N/x]\lambda y.M) & = & \lambda y.([N/x]M) & x \neq y, y \notin FV(N) \\ ([N/x](M\ M')) & = & (([N/x]M)\ ([N/x]M')) & \end{array}$$

In the 4th clause, y should not be a free variable of N This is possible through the use of *α-conversion.* When z is not free in M,

$$(\alpha) \quad \lambda y.M = \lambda z.([z/y]M)$$

In this lecture, we assume that the set of lambda-terms is the quotient set of equivalence classes for (α), i.e., such renaming of bound variables is always allowed.

3 Reduction rules

Definition 2 *λ-calculus is the term rewriting system based on λ-terms, with β-reduction as reduction rules.*

$$(\beta) \quad ((\lambda x.M)\ N) \rightarrow ([N/x]M)$$

Example 2 (β-reduction)

$$\begin{array}{ll} & (\lambda f.\lambda g.\lambda x.f\ x\ (g\ x))\ (\lambda x.\lambda y.x)\ (\lambda x.\lambda y.x) \\ \rightarrow & \lambda x.(\lambda x.\lambda y.x)\ x\ ((\lambda x.\lambda y.x)\ x) \\ \rightarrow & \lambda x.(\lambda y.x)\ (\lambda y.x) \\ \rightarrow & \lambda x.x \end{array}$$

$$\begin{array}{ll} & (\lambda x.x\ x)\ (\lambda x.x\ x) \\ \rightarrow & (\lambda x.x\ x)\ (\lambda x.x\ x) \\ \rightarrow & \ldots \end{array}$$

Theorem 1 (Church-Rosser) *λ-calculus is confluent.* I.e. *When there are 2 reduction sequences $M \to \ldots \to N$ and $M \to \ldots \to P$, then there exists a term T such that $N \to \ldots \to T$ and $P \to \ldots \to T$.*

4 Lambda-calculus is universal

Any program can be written using λ-calculus.

Natural numbers They can be encoded using *Church numerals*

$$\begin{array}{ll} \mathsf{c}_n = \lambda f.\lambda x.(f\ \ldots(f\ x)\ldots) & f \text{ applied } n \text{ times} \\ \mathsf{c}_+ = \lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x) & \text{addition} \\ \mathsf{c}_\times = \lambda m.\lambda n.\lambda f.m\ (n\ f) & \text{multiplication} \end{array}$$

Exercise 1 1. *Compute the normal form of $\mathsf{c}_2\ \mathsf{c}_2$.*

2. *Find a λ-term c_{pow} such that $\mathsf{c}_{pow}\ \mathsf{c}_m\ \mathsf{c}_n$ reduces to c_{m^n}.*

Boole algebra Booleans can be encoded as follows.

$$\mathsf{t} = \lambda x.\lambda y.x \qquad \mathsf{f} = \lambda x.\lambda y.y \qquad \mathsf{not} = \lambda b.\lambda x.\lambda y.b\ y\ x$$

Here is a function that receives a Church numeral as input and returns whether it is equal to 0 or not.

$$\mathsf{if0} = \lambda n.n\ (\lambda x.\mathsf{f})\ \mathsf{t}$$

Cartesian product The cartesion product of two sets can be expressed by encoding pairs, using the following terms:

$$\mathsf{pair} = \lambda x.\lambda y.\lambda f.f\ x\ y \qquad \mathsf{fst} = \lambda p.p\ \mathsf{t} \qquad \mathsf{snd} = \lambda p.p\ \mathsf{f}$$

Here is how it works:

$$\mathsf{fst}\ (\mathsf{pair}\ a\ b) \to \mathsf{pair}\ a\ b\ \mathsf{t} \to (\mathsf{t}\ a\ b) \to a$$

Substraction While multiplication was easy, substraction of Church numerals is comparatively difficult. Here is a possible definition.

$$\begin{array}{lcl} \mathsf{c}_- & = & \lambda m.\lambda n.n\ \mathsf{c}_\mathsf{p}\ m \\ \mathsf{c}_\mathsf{s} & = & \lambda n.\lambda f.\lambda x.f\ (n\ f\ x) \\ \mathsf{s}' & = & \lambda x.\mathsf{pair}\ (\mathsf{snd}\ x)\ (\mathsf{c}_\mathsf{s}\ (\mathsf{snd}\ x)) \\ \mathsf{c}_\mathsf{p} & = & \lambda n.\mathsf{fst}\ (n\ \mathsf{s}'\ (\mathsf{pair}\ \mathsf{c}_0\ \mathsf{c}_0)) \end{array}$$

c_s computes the successor of a number, and c_p its predecessor.

$\mathsf{s}'\ (\mathsf{pair}\ m\ n)$ returns the pair $(\mathsf{c}_\mathsf{s}\ n, m)$. By applying it k times we can obtain the $k-1^{th}$ successor of m.

This property is used by c_p to return the predecessor of n.

Finally, c_- computes the n^{th} predecessor of m by repeatedly applying c_p. If $m \geq n$, then

$$\mathsf{c}_-\ \mathsf{c}_m\ \mathsf{c}_n \overset{*}{\to} \mathsf{c}_{m-n}$$

Alternative encoding of natural numbers The complexity of the definition $\mathsf{c_p}$ is not a problem from a theoretical point of view (what matters here is that one can define it). From that point of view, the ability to use c_n as an iterator is more important.

However one can think of other definitions allowing to structurally extract the predecessor, using the same approach as booleans or pairs. Here is one such example, which we call *analytical*:

$$\begin{array}{rcl} \mathsf{a_0} & = & \lambda s.\lambda z.z \\ \mathsf{a_s} & = & \lambda n.\lambda s.\lambda z.s\ n \\ \mathsf{a_p} & = & \lambda n.n\ (\lambda x.x)\ \mathsf{i_0} \end{array}$$

Here the successor $\mathsf{a_s}$ gives direct access to its predecessor, which allows an easy definition of $\mathsf{a_p}$. However, this definition alone is not sufficient to allow iteration. One could replace it with the fix-point operator we define below, however we will see that this fix-point operator cannot be defined in most typed versions of the lambda-calculus.

The solution is simple: use an *hybrid* version, allowing both extraction of the predecessor, and of an iterator.

$$\begin{array}{rcl} \mathsf{h_0} & = & \mathsf{pair}\ \mathsf{a_0}\ \mathsf{c_0} \\ \mathsf{h_s} & = & \lambda n.\mathsf{pair}\ (\mathsf{a_s}\ n)\ (\mathsf{c_s}\ (\mathsf{snd}\ n)) \\ \mathsf{h_p} & = & \lambda n.n\ (\lambda x.x)\ \mathsf{h_0} \\ \mathsf{h_+} & = & \lambda m.\lambda n.\mathsf{snd}\ m\ \mathsf{h_s}\ n \\ \mathsf{h_-} & = & \lambda m.\lambda n.\mathsf{snd}\ n\ \mathsf{h_p}\ m \\ \mathsf{h_\times} & = & \lambda m.\lambda n.\mathsf{snd}\ m\ (\mathsf{h_+}\ n)\ \mathsf{h_0} \end{array}$$

Since $\mathsf{snd}\ \mathsf{h}_n = \mathsf{c}_n$, it is easy to extract an iterator and use it to define arithmetic operations.

Fix-point operator While the iterator allows us to repeat a function a fixed number of times, this number needs to be computed in advance, so that it cannot be used to compute arbitrary recursive functions.

A general way to define recursive functions is to use the fix-point operator Y. Y is a fix-point operator when $(Y\ M)$ reduces to $(M\ (Y\ M))$, i.e. $(Y\ M)$ is a fix-point of M.

$$Y = (\lambda f.\lambda x.x\ (f\ f\ x))\ (\lambda f.\lambda x.x\ (f\ f\ x))$$

For instance, here is the recursive definition of factorial.

$$\begin{array}{rcl} 0! & = & 1 \\ n! & = & n \times (n-1)! \quad \text{if } n > 0 \end{array}$$

In the syntax of λ-calculus it becomes:

$$\mathsf{c_!} = \lambda n.\mathsf{if0}\ n\ \mathsf{c_1}\ (\mathsf{c_\times}\ n\ (\mathsf{c_!}\ (\mathsf{p}\ n)))$$

Such recursive definitions ($\mathsf{c_!}$ appears in the right-hand side too) are not valid in the λ-calculus itself, but they can be encoded with Y.

$$\mathsf{c_!} = Y(\lambda f.\lambda n.\mathsf{if0}\ n\ \mathsf{c_1}\ (\mathsf{c_\times}\ n\ (f\ (\mathsf{p}\ n)))$$

Since $YM \to M(YM)$, the above equation is valid.

$$\mathsf{c_!} \to (\lambda f.\lambda n.\mathsf{if0}\ n\ \mathsf{c_1}\ (\mathsf{c_\times}\ n\ (f\ (\mathsf{p}\ n))))\ \mathsf{c_!} \to \lambda n.\mathsf{if0}\ n\ \mathsf{c_1}\ (\mathsf{c_\times}\ n\ (\mathsf{c_!}\ (\mathsf{p}\ n)))$$

Exercise 2 *Define $\mathsf{c_!}$ without using Y, i.e. use c_n as iterator.*