Computability and lambda calculus

Jacques Garrigue, 2013.11.12

In 1920, Schönfinkel, a German 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.

4 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 $E ::= \mathcal{R} \mid (E+E) \mid (E-E) \mid (E \times E) \mid (E/E)$

Rewriting rules When both x and y are numbers,

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

Note: $(x+y)$ is a formula, but $x+y$ is the number obtaining by adding x and y.

The above rules are sometimes called *reduction rules.*

Example 1 (rewriting)

$$(15+(1/3)) \times (5-2) \rightarrow (15+0.3333) \times (5-2) \rightarrow (15.3333) \times 3 \rightarrow 46$$

5 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 for another *lambda*-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.

$(M_1\ M_2)$ 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.

Mixing the above grammar with arithmetic,

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

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,

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

Such renaming of bound variables is always allowed.

6 Reduction rules

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

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

Example 2 (β-reduction)

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

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

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 are term T such that $N \to \ldots \to T$ and $P \to \ldots \to T$.*

7 Lambda-calculus is universal

Any program can be written using λ-calculus.

Natural numbers They can be encoded using *Church numerals*

$\mathsf{c}_n = \lambda f.\lambda x.(f\ \ldots (f\ x)\ldots)$	f applied n times
$\mathsf{c}_+ = \lambda m.\lambda n.\lambda f.\lambda x.(m\ f\ (n\ f\ x))$	addition
$\mathsf{c}_\times = \lambda m.\lambda n.\lambda f.(m\ (n\ f))$	multiplication

Exercise 1 *Find the* λ*-term corresponding to exponentiation.*

Boole algebra They 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, subtraction of Church numbers is comparatively difficult. Here is a possible definition.

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

s computes the successor of a number, and$\mathsf{p}$ its predecessor.

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

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

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

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

Fix-point operator In order to define recursive functions, we need the fix-point operator Y. Y is a fix-point operator when $(Y\ M)$ reduces to $(M\ (Y\ M))$.

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

Y is necessary when we don't know hom many times we will need to iterate a function. For instance, here is the recursive definition of factorial.

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

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.

$$\mathsf{c}_! \to (\lambda f.\lambda n.\mathsf{if0}\ n\ \mathsf{c}_1\ (\mathsf{c}_\times\ n\ (f\ (\mathsf{p}\ n))))\ \mathsf{c}_! \to \lambda n.\mathsf{if0}\ n\ \mathsf{c}_1\ (\mathsf{c}_\times\ n\ (\mathsf{c}_!\ (\mathsf{p}\ n)))$$

8 Evaluation strategies

The lambda calculus by itself is not a computer. Evaluation order is not specified, and on some λ-terms evaluation may terminate or not depending on the choice of reductions. The concepts of *normal form* and *strategy* let us define computations more precisely.

Normal form The most logical definition for *normal form* is to require that no *redex* (reducible subterm) be left in a term. But if we want to get closer to the notion of computation, *weak normal form*, where redexes under a λ-abstraction need not be reduced, is more natural. Lazy languages, like Haskell, do not reduce terms in argument position either, so they produce *weak head normal form.*

λ-term	nf	wnf	whnf
$x\ (\lambda y, y),\ \lambda x.\lambda y.x$	$\circ$	$\circ$	$\circ$
$\lambda x.(\lambda y.y)x$	$\times$	$\circ$	$\circ$
$x\ ((\lambda y.y)\ z)$	$\times$	$\times$	$\circ$

Leftmost strategy Reduce leftmost redex first. This amounts to *call-by-name*, *i.e.* functions are called without evaluating their arguments.

$$(\lambda x.x)\ ((\lambda y.y)\ z) \to ((\lambda y.y)\ z)$$

If for some strategy $M \to^* N$ (*i.e.* N has a normal form), the lefmost strategy reaches this normal form.

Rightmost-innermost strategy Reduce the innermost among the rightmost redexes. This amounts to *call-by-value*, *i.e.* function arguments are evaluated before being substitued in the function body.

$$(\lambda x.x)\ ((\lambda y.y)\ z) \to ((\lambda x.x)\ z)$$

If for some strategy $M \to^* \ldots$ (*i.e.* there is an infinite reduction starting for M), then the rightmost-innermost strategy does not terminate.

Abstract machine By combining a definition of normal form with an appropriate strategy, one defines an abstract machine evaluation λ-terms deterministically.

9 Equivalence of Turing machine and Lambda calculus

9.1 Turing machine to lambda term

Tuple and list We have already seen how to build a pair. Using the same principle, one can build n-tuples.

$$(a_1, \ldots a_n)\lambda f.(f\ a_1\ \ldots\ a_n)$$

Variable length sequences, or *lists*, can be encoded using a combination of booleans and pairs. The empty list [] is represented by $\mathsf{f} = \lambda x.\lambda y.y$, and

$$[a_1, a_2, \ldots, a_n] = \lambda x.\lambda y.(x\ (\mathsf{pair}\ a_1\ [a_2, \ldots, a_n]))$$

Infinite lists do not include the empty list, and can be represented by pairs and the fixpoint operator. For instance the infinite repetition of the value a can be represented as

$$[a, a, \ldots] = \mathsf{Y}\ (\lambda x.(\mathsf{pair}\ a\ x))$$

States and symbols We assume that $M = (K, \Sigma, q_0, H, \delta)$, $|K| = k$ and $|\Sigma| = l$. The sets of states and symbols can be enumerated, so that $K = \{q_0, \ldots, q_{k-1}\}$ and $\Sigma = \{\sigma_0 = \mathrm{B}, \ldots, \sigma_{l-1}\}$. We can the use the following translation:

$$\overline{q_i} = \lambda x_0 \ldots x_{k-1}.x_i$$
$$\overline{\sigma_i} = \lambda x_0 \ldots x_{l-1}.x_i$$

Global state A global state (T, n, q) can be represented as the 4-uple

$$(\overline{q}, \overline{T(n)}, [\overline{T(n-1)}, \overline{T(n-2)}, \ldots], [\overline{T(n+1)}, \overline{T(n+2)}, \ldots])$$

Here we use an infinite list, but it ends with an infinite repetition of B's.

$$[\overline{\mathrm{B}}, \ldots] = Y(\lambda y.(\overline{\mathrm{B}}, y)) \to^* (\overline{\mathrm{B}}, Y(\lambda y.(\overline{\mathrm{B}}, y))) \to^* \ldots$$

Transition function The transition function between global states is represented by a k-uple of l-uples of step-functions of the following form:

$$\Delta = \lambda t.t((\overline{\delta}(q_0, \sigma_0), \overline{\delta}(q_0, \sigma_1), \ldots, \overline{\delta}(q_0, \sigma_{l-1})), \ldots, (\overline{\delta}(q_{k-1}, \sigma_0), \ldots))$$

$$\overline{\delta}(q, \sigma) = \begin{cases} \lambda t_l.\lambda t_r.f(\overline{q'}, \mathsf{fst}\ t_l, \mathsf{snd}\ t_l, (\overline{\sigma'}, t_r)) & \text{when } \delta(q, \sigma) = (\sigma', \leftarrow, q') \\ \lambda t_l.\lambda t_r.f(\overline{q'}, \mathsf{fst}\ t_r, (\overline{\sigma'}, t_l), \mathsf{snd}\ t_r) & \text{when } \delta(q, \sigma) = (\sigma', \rightarrow, q') \\ \lambda t_l.\lambda t_r.(\overline{q}, \overline{\sigma}, t_l, t_r) & \text{when } q \in H \end{cases}$$

If we replace here f by the identity function $\lambda x.x$, we can see Δ as a function from global to global state:

$$\Delta : (K, \Sigma, \Sigma\ \mathit{list}, \Sigma\ \mathit{list}) \to (K, \Sigma, \Sigma\ \mathit{list}, \Sigma\ \mathit{list})$$

Execution Δ only runs the Turing machine 1 step. To go further, we need to use the fixed-point operator. We do this by abstracting on f.

$$\begin{aligned} \lambda f.\Delta :\ & ((K, \Sigma, \Sigma\ \mathit{list}, \Sigma\ \mathit{list}) \to (K, \Sigma, \Sigma\ \mathit{list}, \Sigma\ \mathit{list})) \to \\ & ((K, \Sigma, \Sigma\ \mathit{list}, \Sigma\ \mathit{list}) \to (K, \Sigma, \Sigma\ \mathit{list}, \Sigma\ \mathit{list})) \end{aligned}$$

$$(Y\ (\lambda f.\Delta)) : \quad (K, \Sigma, \Sigma\ \mathit{list}, \Sigma\ \mathit{list}) \to (K, \Sigma, \Sigma\ \mathit{list}, \Sigma\ \mathit{list})$$

By applying this function to the initial state T we obtain a computation $T \rhd (T', n, q')$. Since we use Y both for the transition function and for the tape, we must be careful about only reducing to weak-head normal form.

$$(Y\ (\lambda f.\Delta))\ (\overline{q_0}, \overline{T(0)}, [\overline{T(-1)}, \ldots], [\overline{T(1)}, \ldots]) \to^* (\overline{q'}, \overline{T'(n)}, [\overline{T'(n-1)}, \ldots], [\overline{T'(n+1)}, \ldots])$$

9.2 Turing machine interpreting lambda-terms

In this section we consider the opposite direction, where we write a lambda-term on a tape, and use a Turing machine to reduce it.

Before thinking about the Turing machine itself, it is important to represent lambda-terms in a way that makes their evaluation easy. In particular, we wish to avoid α-conversion. De Bruijn indices allows us to do that.

Definition 3 De Bruijn indices *The unnamed lambda-calculus is defined as follows.*

$$M ::= n \mid \lambda M \mid (M\ M)$$

Indices n represent variables. The n indicates the position of the λ binding this variable, starting from inside. For instance $\lambda x.\lambda y.x = \lambda\lambda 2$, $\lambda x.(x\ x) = \lambda(1\ 1)$.

We need to redefine substitution

$$\begin{array}{lllllll}
\Uparrow_n k & = & k+1 & k \geq n & [N/n]n & = & N \\
\Uparrow_n k & = & k & k < n & [N/n]k & = & k-1 \quad k > n \\
\Uparrow_n \lambda M & = & \lambda(\Uparrow_{n+1} M) & & [N/n]k & = & k \quad k < n \\
 & & & & [N/n]\lambda M & = & \lambda([\Uparrow_n N/n+1]M) \\
\beta & & \lambda M\ N \to [N/1]M & & [N/n](M\ M') & = & ([N/n]M\ [N/n]M')
\end{array}$$

Example 3 *We extract the 1st element of a pair.*

$$\mathsf{fst}\ \lambda(1\ a\ b) = \lambda(1\ \lambda\lambda 2)\ \lambda(1\ a\ b) \to \lambda(1\ a\ b)\ \lambda\lambda 2 \to \lambda\lambda 2\ a\ b \to \lambda a\ b \to a$$

Format of the tape We use 6 symbols $\{\mathrm{B}, 0, 1, \mathrm{E}, \lambda, @\}$, and translate a lambda-term M into a tape $\overline{M}$ in the following way.

$$\begin{array}{rcl}
\overline{n} & = & (n\text{'s binary representation})\ \mathrm{E} \\
\overline{\lambda M} & = & \lambda\ \overline{M} \\
\overline{(M\ M')} & = & @\ \overline{M}\ \overline{M'}
\end{array}$$

Turing machine Λ We do not give here the definition of the Turing machine Λ, but it is clear that it can be defined using operations such as copying and addition/substraction. Here we use the leftmost strategy to weak-head normal form.

9.3 Universal Turing machine

We can represent any Turing machine as a lambda-term, and we can convert this lambda-term to a tape evaluated by the Turing machine Λ. This give us a simple proof of the existence of universal Turing machines.

Corollary 1 *There exists a universal Turing machine, such that it is able to simulate any other Turing machine, by being given its description as input.*

Proof We just have to translate the simulated Turing maching and its input tape to a lambda-term, and feed it as input to Λ.