Lambda calculus: Syntax

Jacques Garrigue, 2009.5.14

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

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)

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

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 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,

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

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,

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

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 α-conversion and 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 $M \rightarrow \ldots \rightarrow N$ and $M \rightarrow \ldots \rightarrow P$, then there exists are term T such that $N \rightarrow \ldots \rightarrow T$ and $P \rightarrow \ldots \rightarrow T$.*

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 *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.

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

Subtraction While multiplication was easy, subtraction of Church numbers is comparatively difficult. Here is a possible definition.

$$\begin{array}{lcl} \mathsf{c}_- & = & \lambda m.\lambda n.(n\ \mathsf{p}\ m) \\ \mathsf{s} & = & \lambda n.\lambda f.\lambda x.(f\ (n\ f\ x)) \\ \mathsf{p} & = & \lambda n.(n\ (\lambda z.(\mathsf{ifn}\ z\ (\mathsf{s}\ z)\ \mathsf{f}))\ \mathsf{t}) \\ \mathsf{ifn} & = & \lambda z.(z\ (\lambda x.x)\ (\lambda x.\mathsf{t})\ \mathsf{f}) \end{array}$$

s computes the successor of a number, andp its predecessor. p requires and auxiliary function ifn.

ifn takes either Church numeral c_k or t as input, and returns t in the 1st case, and f in the 2nd case. ifn is a function from Church numerals and booleans to booleans.

$$\begin{array}{rcl} \mathsf{i} & = & \lambda x.x \\ \mathsf{ifn}\ \mathsf{c}_k & \to & (\mathsf{c}_k\ \mathsf{i}\ (\lambda x.\mathsf{t}))\ \mathsf{f} \\ & \to & (\mathsf{i}\ldots(\mathsf{i}\ (\lambda x.\mathsf{t}))\ldots)\ \mathsf{f} \\ & \to & (\lambda x.\mathsf{t})\ \mathsf{f} \\ & \to & \mathsf{t} \\ \mathsf{ifn}\ \mathsf{t} & \to & (\mathsf{t}\ \mathsf{i}\ (\lambda x.\mathsf{t}))\ \mathsf{f} \\ & \to & \mathsf{i}\ \mathsf{f} \\ & \to & \mathsf{f} \end{array}$$

Now $\mathsf{s}' = \lambda z.(\mathsf{ifn}\ z\ (\mathsf{s}\ z)\ \mathsf{f})$ (as in the definition of p) is such that it maps t to $\mathsf{f} = \mathsf{c}_0$, and c_k to c_{k+1}. As result, p c_k, applying s' k times to t, will return exactly c_{k-1}.

To compute $m - n$, we must apply p n times to m. When $m \geq n$,

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

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.

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

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.

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

5 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.