Lecture 37

The λ-Calculus

The λ-calculus (λ = "lambda," Greek for l) consists of a set of objects called λ-*terms* and some rules for manipulating them. It was originally designed to capture formally the notions of ***functional abstraction*** and ***functional application*** and their interaction.

The λ-calculus has had a profound impact on computing. One can see the basic principles of the λ-calculus at work in the functional programming language LISP and its more modern offspring SCHEME and DYLAN.

In mathematics, λ-notation is commonly used to represent functions. The expression $\lambda x.E(x)$ denotes a function that on input x computes $E(x)$. To apply this function to an input, one substitutes the input for the variable x in the body $E(x)$ and evaluates the resulting expression.

For example, the expression

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

might be used to denote the successor function on natural numbers. To apply this function to the input 7, we would substitute 7 for x in the body and evaluate:

$$(\lambda x.(x+1))7 \rightarrow 7+1=8.$$

In the programming language DYLAN, one would write

```
(method (x) (+ x 1))
```

for the same thing. The keyword `method` is really λ in disguise. If you typed

```
((method (x) (+ x 1)) 7)
```

at a DYLAN interpreter, it would print out 8.

For another example, the expression

$$\lambda x.f(gx)$$

denotes the composition of the functions f and g; that is, the function that on input x applies g to x, then applies f to the result. The expression

$$\lambda f.\lambda g.\lambda x.f(gx) \tag{37.1}$$

denotes the function that takes functions f and g as input and gives back their composition $\lambda x.f(gx)$. In DYLAN one would write

```
(method (f)
   (method (g)
      (method (x) (f (g x)))))
```

To see how this works, let's apply (37.1) to the successor function twice. We use different variables in the successor functions below for clarity. The symbol $\rightarrow$ denotes one substitution step.

$(\lambda f.\lambda g.\lambda x.(f(gx)))\,(\lambda y.(y+1))\,(\lambda z.(z+1))$	substitute $\lambda y.(y+1)$ for f
$\rightarrow (\lambda g.\lambda x.((\lambda y.(y+1))(gx)))\,(\lambda z.(z+1))$	substitute $\lambda z.(z+1)$ for g
$\rightarrow \lambda x.((\lambda y.(y+1))\,((\lambda z.(z+1))x))$	substitute x for z
$\rightarrow \lambda x.((\lambda y.(y+1))\,(x+1))$	substitute $x+1$ for y
$\rightarrow \lambda x.((x+1)+1)$	

We could have substituted gx for y in the second step or $(\lambda z.(z+1))x$ for y in the third; we would have arrived at the same final result.

Functions represented by λ-terms have only one input. A function with two inputs x, y that returns a value M is modeled by a function with one input x that returns a function with one input y that returns a value M. The technical term for this trick is *currying* (after Haskell B. Curry).

The Pure λ-Calculus

In the *pure* λ-calculus, there are only variables $\{f, g, h, x, y, \ldots\}$ and operators for λ-abstraction and application. Syntactic objects called λ-*terms* are built inductively from these:

- any variable x is a λ-term;
- if M and N are λ-terms, then MN is a λ-term (functional application—think of M as a function that is about to be applied to input N); and
- if M is a λ-term and x is a variable, then $\lambda x.M$ is a λ-term (functional abstraction—think of $\lambda x.M$ as the function that on input x computes M).

The operation of application is not associative, and unparenthesized expressions are conventionally associated to the left; thus, MNP should be parsed $(MN)P$.

In the pure λ-calculus, λ-terms serve as both functions and data. There is nothing like "+1" as we used it informally above, unless we encode it somehow. We'll show how to do this below.

The substitution rule described informally above is called *β-reduction.* Formally, this works as follows. Whenever our λ-term contains a subterm of the form $(\lambda x.M)N$, we can replace this subterm by the term $\mathrm{s}^x_N(M)$, where $\mathrm{s}^x_N(M)$ denotes the term obtained by

(i) renaming the bound variables of M (those y occurring in the scope of some λy) as necessary so that neither x nor any variable of N occurs bound in M; and

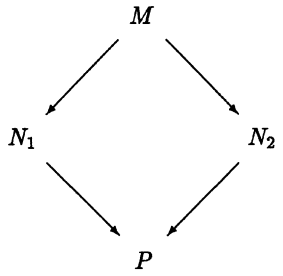
(ii) substituting N for all occurrences of x in the resulting term.

Step (i) is necessary only to make sure that any free variables y of N will not be inadvertently captured by a λy occurring in M when the substitution is done in step (ii). This is the same problem that comes up in first-order logic. We can rename bound variables in λ-terms anytime, since their behavior as functions is not changed. For example, we can rewrite $\lambda y.xy$ as $\lambda z.xz$; intuitively, the function that on input y applies x to y is the same as the function that on input z applies x to z. The process of renaming bound variables is officially called *α-reduction.*

We denote α- and β-reduction by $\xrightarrow{\alpha}$ and $\xrightarrow{\beta}$, respectively. Thus

$$(\lambda x.M)N \xrightarrow{\beta} \mathrm{s}^x_N(M).$$

Computation in the λ-calculus is performed by β-reducing subterms whenever possible and for as long as possible. The order of the reductions doesn't matter, since there is a theorem that says that if you can reduce M to N_1 by some sequence of reduction steps and M to N_2 by some other sequence of reduction steps, then there exists a term P such that both N_1 and N_2 reduce to P.



This is called the *Church–Rosser property* after Alonzo Church and J. Barkley Rosser.

A term is said to be in *normal form* if no β-reductions apply; that is, if it has no subterms of the form $(\lambda x.M)N$. A normal form corresponds roughly to a halting configuration of a Turing machine. By the Church–Rosser property, if a λ-term has a normal form, then that normal form is unique up to α-renaming.

There are terms with no normal form. These correspond to nonhalting computations of Turing machines. For example, the λ-term

$$(\lambda x.xx)(\lambda x.xx)$$

has no normal form—try to do a β-reduction and see what happens! The term $\lambda x.xx$ is analogous to a Turing machine that on input x runs M_x on x.

Church Numerals

To simulate the μ-recursive functions in the λ-calculus, we must first encode the natural numbers as λ-terms so they can be used in computations. Alonzo Church came up with a nice way to do this. His encoding is known as the *Church numerals*:

$$\begin{aligned}
\overline{0} &\stackrel{\text{def}}{=} \lambda f.\lambda x.x,\\
\overline{1} &\stackrel{\text{def}}{=} \lambda f.\lambda x.fx,\\
\overline{2} &\stackrel{\text{def}}{=} \lambda f.\lambda x.f(fx),\\
\overline{3} &\stackrel{\text{def}}{=} \lambda f.\lambda x.f(f(fx)),\\
&\vdots\\
\overline{n} &\stackrel{\text{def}}{=} \lambda f.\lambda x.f^n x,\\
&\vdots
\end{aligned}$$

where $f^n x$ is an abbreviation for the term

$$\underbrace{f(f(\cdots(f}_{n} x)\cdots)).$$

In other words, $\overline{n}$ represents a function that on input f returns the n-fold composition of f with itself. The $\overline{n}$ are all distinct and in normal form.

Using this representation of the natural numbers, the successor function can be defined as

$$s \stackrel{\text{def}}{=} \lambda m.\lambda f.\lambda x.f(mfx).$$

To see that this is correct, try applying it to any $\overline{n}$:

$$\begin{aligned}
s\overline{n} &= (\lambda m.\lambda f.\lambda x.f(mfx))\,(\lambda f.\lambda x.f^n x) \\
&\xrightarrow{\alpha} (\lambda m.\lambda g.\lambda y.g(mgy))\,(\lambda f.\lambda x.f^n x) \\
&\xrightarrow{\beta} \lambda g.\lambda y.g((\lambda f.\lambda x.f^n x)gy) \\
&\xrightarrow{\beta} \lambda g.\lambda y.g((\lambda x.g^n x)y) \\
&\xrightarrow{\beta} \lambda g.\lambda y.g(g^n y) \\
&= \lambda g.\lambda y.g^{n+1} y \\
&\xrightarrow{\alpha} \lambda f.\lambda x.f^{n+1} x \\
&= \overline{n+1}.
\end{aligned}$$

One can likewise define addition, multiplication, and all the other μ-recursive functions.

Combinatory Logic

Combinatory logic is a form of variable-free λ-calculus. It was first invented to study the mathematics of symbol manipulation, especially substitution. The system consists of terms called *combinators* that are manipulated using *reduction rules.*

There are two primitive combinators S and K, which are just symbols, as well as a countable set of variables $\{X, Y, \ldots\}$. More complicated combinators are formed inductively: S, K, and variables are combinators; and if M and N are combinators, then so is MN. Here MN is just a term, a syntactic object, but we can think of M as a function and N as its input; thus, MN represents the application of M to N. As with the λ-calculus, this operation is not associative, so we use parentheses to avoid ambiguity. By convention, a string of applications associates to the left; thus, XYZ should be parsed $(XY)Z$ and not $X(YZ)$.

Computation proceeds according to two reduction rules, one for S and one for K. For any terms M, N, and P,

$$SMNP \to MP(NP),$$
$$KMN \to M.$$

Computation in this system consists of a sequence of reduction steps applied to subterms of a term.

Other combinators can be built from S and K. For example, the combinator $I \stackrel{\text{def}}{=} SKK$ acts as the identity function: for any X,

$$\begin{aligned} IX &= SKKX \\ &\to KX(KX) && \text{the } S \text{ rule} \\ &\to X && \text{the } K \text{ rule.} \end{aligned}$$

Let $B = SK$. Whereas K picks out the first element of a pair, B picks out the second element:

$$BXY = SKXY \to KY(XY) \to Y.$$

One can construct fancy combinators from S and K that can rearrange symbols in every conceivable way. For example, to take two inputs and apply the second to the first, use the combinator $C = S(S(KS)B)K$:

$$\begin{aligned} CXY &= S(S(KS)B)KXY \\ &\to S(KS)BX(KX)Y \\ &\to KSX(BX)(KX)Y \\ &\to S(BX)(KX)Y \\ &\to BXY(KXY) \\ &\to YX. \end{aligned}$$

There is a theorem that says that no matter how you want to rearrange your inputs, there is a combinator built from S and K only that can do it. In other words, for any term M built from $X_1, \ldots, X_n$ and the application operator, there is a combinator D built from S and K only such that

$$DX_1X_2 \cdots X_n \stackrel{*}{\longrightarrow} M.$$

This theorem is called *combinatorial completeness.*

There is a *paradoxical combinator* $SII(SII)$, which corresponds to the λ-term $(\lambda x.xx)(\lambda x.xx)$. Like its counterpart, it has no normal form.

Like the λ-calculus, combinatory logic is powerful enough to simulate Turing machines.

Historical Notes

The late 1920s and 1930s were a hectic time. Turing machines (Turing [120]), Post systems (Post [99, 100]), μ-recursive functions (Gödel [51], Herbrand, Kleene [67]), the λ-calculus (Church [23, 24, 25, 26], Kleene [66], Rosser [107]), and combinatory logic (Schönfinkel [111], Curry [29]) were all developed around this time.

The λ-calculus is a topic unto itself. Barendregt's book [9] is an indispensable reference.

The μ-recursive functions were formulated by Gödel and presented in a series of lectures at Princeton in 1934. According to Church [25], Gödel acknowledged that he got the idea originally from Jacques Herbrand in conversation.

A proof of the equivalence of the μ-recursive functions and the λ-calculus first appeared in Church [25], although Church attributes the proof chiefly to Kleene. The equivalence of TMs and the λ-calculus was shown by Turing [120].

Various perspectives on this important period can be found in Kleene [69], Davis [31, 32], Rogers [106], Yasuhara [124], Jones [63], Brainerd and Landweber [15], Hennie [58], and Machtey and Young [81].

Chomsky [18] defined the Chomsky hierarchy and proved that the type 0 grammars generate exactly the r.e. sets.

The relationship between context-sensitive grammars and linear bounded automata was studied by Myhill [92], Landweber [78], and Kuroda [77].