Chapter 2

Lambda calculus

Lambda calculus is based on the so-called 'lambda notation' for denoting functions. In informal mathematics, when one wants to refer to a function, one usually first gives the function an arbitrary name, and thereafter uses that name, e.g.

> Suppose $f : \mathbb{R} \to \mathbb{R}$ is defined by:
>
> $$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ x^2 sin(1/x^2) & \text{if } x \neq 0 \end{cases}$$
>
> Then $f'(x)$ is not Lebesgue integrable over the unit interval $[0, 1]$.

Most programming languages, C for example, are similar in this respect: we can define functions only by giving them names. For example, in order to use the successor function (which adds 1 to its argument) in nontrivial ways (e.g. consider a pointer to it), then even though it is very simple, we need to name it via some function definition such as:

```
int suc(int n)
 { return n + 1;
 }
```

In either mathematics or programming, this seems quite natural, and generally works well enough. However it can get clumsy when higher order functions (functions that manipulate other functions) are involved. In any case, if we want to treat functions on a par with other mathematical objects, the insistence on naming is rather inconsistent. When discussing an arithmetical expression built up from simpler ones, we just write the subexpressions down, without needing to give them names. Imagine if we always had to deal with arithmetic expressions in this way:

> Define x and y by $x = 2$ and $y = 4$ respectively. Then $xx = y$.

Lambda notation allows one to denote functions in much the same way as any other sort of mathematical object. There is a mainstream notation sometimes used in mathematics for this purpose, though it's normally still used as part of the definition of a temporary name. We can write

$$x \mapsto t[x]$$

to denote the function mapping any argument x to some arbitrary expression $t[x]$, which usually, but not necessarily, contains x (it is occasionally useful to "throw away" an argument). However, we shall use a different notation developed by Church (1941):

$$\lambda x.\ t[x]$$

which should be read in the same way. For example, $\lambda x.x$ is the identity function which simply returns its argument, while $\lambda x.\ x^2$ is the squaring function.

The symbol λ is completely arbitrary, and no significance should be read into it. (Indeed one often sees, particularly in French texts, the alternative notation $[x]\ t[x]$.) Apparently it arose by a complicated process of evolution. Originally, the famous *Principia Mathematica* (Whitehead and Russell 1910) used the 'hat' notation $t[\hat{x}]$ for the function of x yielding $t[x]$. Church modified this to $\hat{x}.\ t[x]$, but since the typesetter could not place the hat on top of the x, this appeared as $\wedge x.\ t[x]$, which then mutated into $\lambda x.\ t[x]$ in the hands of another typesetter.

2.1 The benefits of lambda notation

Using lambda notation we can clear up some of the confusion engendered by informal mathematical notation. For example, it's common to talk sloppily about '$f(x)$', leaving context to determine whether we mean f itself, or the result of applying it to particular x. A further benefit is that lambda notation gives an attractive analysis of practically the whole of mathematical notation. If we start with variables and constants, and build up expressions using just lambda-abstraction and application of functions to arguments, we can represent very complicated mathematical expressions.

We will use the conventional notation $f(x)$ for the application of a function f to an argument x, except that, as is traditional in lambda notation, the brackets may be omitted, allowing us to write just $f\ x$. For reasons that will become clear in the next paragraph, we assume that function application associates to the left, i.e. $f\ x\ y$ means $(f(x))(y)$. As a shorthand for $\lambda x.\ \lambda y.\ t[x,y]$ we will use $\lambda x\ y.\ t[x,y]$, and so on. We also assume that the scope of a lambda abstraction extends as far to the right as possible. For example $\lambda x.x\ y$ means $\lambda x.(x\ y)$ rather than $(\lambda x.\ x)\ y$.

At first sight, we need some special notation for functions of several arguments. However there is a way of breaking down such applications into ordinary lambda notation, called *currying*, after the logician Curry (1930). (Actually the device had previously been used by both Frege (1893) and Schönfinkel (1924), but it's easy to understand why the corresponding appellations haven't caught the public imagination.) The idea is to use expressions like $\lambda x\ y.\ x + y$. This may be regarded as a function $\mathbb{R} \to (\mathbb{R} \to \mathbb{R})$, so it is said to be a 'higher order function' or 'functional' since when applied to one argument, it yields another function, which then accepts the second argument. In a sense, it takes its arguments one at a time rather than both together. So we have for example:

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

Observe that function application is assumed to associate to the left in lambda notation precisely because currying is used so much.

Lambda notation is particularly helpful in providing a unified treatment of bound variables. Variables in mathematics normally express the dependency of some expression on the value of that variable; for example, the value of $x^2 + 2$ depends on the value of x. In such contexts, we will say that a variable is *free*. However there are other situations where a variable is merely used as a place-marker, and does not indicate such a dependency. Two common examples are the variable m in

$$\sum_{m=1}^{n} m = \frac{n(n+1)}{2}$$

and the variable y in

$$\int_0^x 2y + a\ dy = x^2 + ax$$

In logic, the quantifiers $\forall x.\ P[x]$ ('for all x, $P[x]$') and $\exists x.\ P[x]$ ('there exists an x such that $P[x]$') provide further examples, and in set theory we have set abstractions like $\{x \mid P[x]\}$ as well as indexed unions and intersections. In such cases, a variable is said to be *bound*. In a certain subexpression it is free, but in the whole expression, it is bound by a *variable-binding operation* like summation. The part 'inside' this variable-binding operation is called the *scope* of the bound variable.

A similar situation occurs in most programming languages, at least from Algol 60 onwards. Variables have a definite scope, and the formal arguments of procedures and functions are effectively bound variables, e.g. n in the C definition of the successor function given above. One can actually regard variable declarations as binding operations for the enclosed instances of the corresponding variable(s). Note, by the way, that the *scope* of a variable should be distinguished sharply from its *lifetime*. In the C function `rand` that we gave in the introduction, n had a textually limited scope but it retained its value even outside the execution of that part of the code.

We can freely change the name of a bound variable without changing the meaning of the expression, e.g.

$$\int_0^x 2z + a\ dz = x^2 + ax$$

Similarly, in lambda notation, $\lambda x.\ E[x]$ and $\lambda y.\ E[y]$ are equivalent; this is called *alpha*-equivalence and the process of transforming between such pairs is called alpha-conversion. We should add the proviso that y is not a free variable in $E[x]$, or the meaning clearly may change, just as

$$\int_0^x 2a + a\ da \neq x^2 + ax$$

It is possible to have identically-named free and bound variables in the same expression; though this can be confusing, it is technically unambiguous, e.g.

$$\int_0^x 2x + a\ dx = x^2 + ax$$

In fact the usual Leibniz notation for derivatives has just this property, e.g. in:

$$\frac{d}{dx}x^2 = 2x$$

x is used both as a bound variable to indicate that differentiation is to take place with respect to x, and as a free variable to show where to evaluate the resulting derivative. This can be confusing; e.g. $f'(g(x))$ is usually taken to mean something different from $\frac{d}{dx}f(g(x))$. Careful writers, especially in multivariate work, often make the separation explicit by writing:

$$|\frac{d}{dx}x^2|_x = 2x$$

or

$$|\frac{d}{dz}z^2|_x = 2x$$

Part of the appeal of lambda notation is that all variable-binding operations like summation, differentiation and integration can be regarded as functions applied to lambda-expressions. Subsuming all variable-binding operations by lambda abstraction allows us to concentrate on the technical problems of bound variables in one particular situation. For example, we can view $\frac{d}{dx}x^2$ as a syntactic sugaring of $D\ (\lambda x.\ x^2)\ x$ where $D : (\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to \mathbb{R}$ is a differentiation operator, yielding the derivative of its first (function) argument at the point indicated by its second argument. Breaking down the everyday syntax completely into lambda notation, we have $D\ (\lambda x.\ \mathrm{EXP}\ x\ 2)\ x$ for some constant EXP representing the exponential function.

In this way, lambda notation is an attractively general 'abstract syntax' for mathematics; all we need is the appropriate stock of constants to start with. Lambda abstraction seems, in retrospect, to be the appropriate primitive in terms of which to analyze variable binding. This idea goes back to Church's encoding of higher order logic in lambda notation, and as we shall see in the next chapter, Landin has pointed out how many constructs from programming languages have a similar interpretation. In recent times, the idea of using lambda notation as a universal abstract syntax has been put especially clearly by Martin-Löf, and is often referred to in some circles as 'Martin-Löf's theory of expressions and arities'.[1]

2.2 Russell's paradox

As we have said, one of the appeals of lambda notation is that it permits an analysis of more or less all of mathematical syntax. Originally, Church hoped to go further and include set theory, which, as is well known, is powerful enough to form a foundation for much of modern mathematics. Given any set S, we can form its so-called *characteristic predicate* χ_S, such that:

$$\chi_S(x) = \begin{cases} true & \text{if } x \in S \\ false & \text{if } x \notin S \end{cases}$$

Conversely, given any unary predicate (i.e. function of one argument) P, we can consider the set of all x satisfying $P(x)$ — we will just write $P(x)$ for $P(x) = true$. Thus, we see that sets and predicates are just different ways of talking about the same thing. Instead of regarding S as a set, and writing $x \in S$, we can regard it as a predicate and write $S(x)$.

This permits a natural analysis into lambda notation: we can allow arbitrary lambda expressions as functions, and hence indirectly as sets. Unfortunately, this turns out to be inconsistent. The simplest way to see this is to consider the Russell paradox of the set of all sets that do not contain themselves:

$$R = \{x \mid x \notin x\}$$

We have $R \in R \Leftrightarrow R \notin R$, a stark contradiction. In terms of lambda defined functions, we set $R = \lambda x. \neg(x\ x)$, and find that $R\ R = \neg(R\ R)$, obviously counter to the intuitive meaning of the negation operator $\neg$.

To avoid such paradoxes, Church (1940) followed Russell in augmenting lambda notation with a notion of *type*; we shall consider this in a later chapter. However the paradox itself is suggestive of some interesting possibilities in the standard, untyped, system, as we shall see later.

[1] This was presented at the Brouwer Symposium in 1981, but was not described in the printed proceedings.

2.3 Lambda calculus as a formal system

We have taken for granted certain obvious facts, e.g. that $(\lambda y.\, 1 + y)\; 2 = 1 + 2$, since these reflect the intended meaning of abstraction and application, which are in a sense converse operations. Lambda *calculus* arises if we enshrine certain such principles, and *only* those, as a set of formal rules. The appeal of this is that the rules can then be used mechanically, just as one might transform $x - 3 = 5 - x$ into $2x = 5 + 3$ without pausing each time to think about *why* these rules about moving things from one side of the equation to the other are valid. As Whitehead (1919) says, symbolism and formal rules of manipulation:

> [...] have invariably been introduced to make things easy. [...] by the aid of symbolism, we can make transitions in reasoning almost mechanically by the eye, which otherwise would call into play the higher faculties of the brain. [...] Civilisation advances by extending the number of important operations which can be performed without thinking about them.

2.3.1 Lambda terms

Lambda calculus is based on a formal notion of lambda term, and these terms are built up from variables and some fixed set of constants using the operations of function application and lambda abstraction. This means that every lambda term falls into one of the following four categories:

1. **Variables:** these are indexed by arbitrary alphanumeric strings; typically we will use single letters from towards the end of the alphabet, e.g. x, y and z.

2. **Constants:** how many constants there are in a given syntax of lambda terms depends on context. Sometimes there are none at all. We will also denote them by alphanumeric strings, leaving context to determine when they are meant to be constants.

3. **Combinations**, i.e. the application of a function s to an argument t; both these components s and t may themselves be arbitrary λ-terms. We will write combinations simply as $s\ t$. We often refer to s as the 'rator' and t as the 'rand' (short for 'operator' and 'operand' respectively).

4. **Abstractions** of an arbitrary lambda-term s over a variable x (which may or may not occur free in s), denoted by $\lambda x.\, s$.

Formally, this defines the set of lambda terms inductively, i.e. lambda terms arise *only* in these four ways. This justifies our:

- Defining functions over lambda terms by primitive recursion.
- Proving properties of lambda terms by structural induction.

A formal discussion of inductive generation, and the notions of primitive recursion and structural induction, may be found elsewhere. We hope most readers who are unfamiliar with these terms will find that the examples below give them a sufficient grasp of the basic ideas.

We can describe the syntax of lambda terms by a BNF (Backus-Naur form) grammar, just as we do for programming languages.

$$Exp = Var \mid Const \mid Exp\ Exp \mid \lambda\, Var.\ Exp$$

and, following the usual computer science view, we will identify lambda terms with abstract syntax trees, rather than with sequences of characters. This means that conventions such as the left-association of function application, the reading of $\lambda x\ y.\ s$ as $\lambda x.\ \lambda y.\ s$, and the ambiguity over constant and variable names are purely a matter of parsing and printing for human convenience, and not part of the formal system.

One feature worth mentioning is that we use single characters to stand for variables and constants in the formal system of lambda terms and as so-called 'metavariables' standing for arbitrary terms. For example, $\lambda x.\ s$ might represent the constant function with value s, or an arbitrary lambda abstraction using the variable x. To make this less confusing, we will normally use letters such as s, t and u for metavariables over terms. It would be more precise if we denoted the variable x by V_x (the x'th variable) and likewise the constant k by C_k — then all the variables in terms would have the same status. However this makes the resulting terms a bit cluttered.

2.3.2 Free and bound variables

We now formalize the intuitive idea of free and bound variables in a term, which, incidentally, gives a good illustration of defining a function by primitive recursion. Intuitively, a variable in a term is free if it does not occur inside the scope of a corresponding abstraction. We will denote the set of free variables in a term s by $FV(s)$, and define it by recursion as follows:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(c) &= \emptyset \\ FV(s\ t) &= FV(s) \cup FV(t) \\ FV(\lambda x.\ s) &= FV(s) - \{x\} \end{aligned}$$

Similarly we can define the set of bound variables in a term $BV(s)$:

$$\begin{aligned} BV(x) &= \emptyset \\ BV(c) &= \emptyset \\ BV(s\ t) &= BV(s) \cup BV(t) \\ BV(\lambda x.\ s) &= BV(s) \cup \{x\} \end{aligned}$$

For example, if $s = (\lambda x\ y.\ x)\ (\lambda x.\ z\ x)$ we have $FV(s) = \{z\}$ and $BV(s) = \{x, y\}$. Note that in general a variable can be both free and bound in the same term, as illustrated by some of the mathematical examples earlier. As an example of using structural induction to establish properties of lambda terms, we will prove the following theorem (a similar proof works for BV too):

Theorem 2.1 *For any lambda term s, the set $FV(s)$ is finite.*
Proof: *By structural induction. Certainly if s is a variable or a constant, then by definition $FV(s)$ is finite, since it is either a singleton or empty. If s is a combination $t\ u$ then by the inductive hypothesis, $FV(t)$ and $FV(u)$ are both finite, and then $FV(s) = FV(t) \cup FV(u)$, which is therefore also finite (the union of two finite sets is finite). Finally, if s is of the form $\lambda x.\, t$ then $FV(t)$ is finite, by the inductive hypothesis, and by definition $FV(s) = FV(t) - \{x\}$ which must be finite too, since it is no larger. Q.E.D.*

2.3.3 Substitution

The rules we want to formalize include the stipulation that lambda abstraction and function application are inverse operations. That is, if we take a term $\lambda x.\ s$ and apply it as a function to an argument term t, the answer is the term s with all free instances of x replaced by t. We often make this more transparent in discussions by using the notation $\lambda x.\ s[x]$ and $s[t]$ for the respective terms.

However this simple-looking notion of substituting one term for a variable in another term is surprisingly difficult. Some notable logicians have made faulty statements regarding substitution. In fact, this difficulty is rather unfortunate since as we have said, the appeal of formal rules is that they can be applied mechanically.

We will denote the operation of substituting a term s for a variable x in another term t by $t[s/x]$. One sometimes sees various other notations, e.g. $t[x{:=}s]$, $[s/x]t$, or even $t[x/s]$. The notation we use is perhaps most easily remembered by noting the vague analogy with multiplication of fractions: $x[t/x] = t$. At first sight, one can define substitution formally by recursion as follows:

$$\begin{aligned} x[t/x] &= t \\ y[t/x] &= y \text{ if } x \neq y \end{aligned}$$

$$\begin{array}{rcl} c[t/x] & = & c \\ (s_1\ s_2)[t/x] & = & s_1[t/x]\ s_2[t/x] \\ (\lambda x.\ s)[t/x] & = & \lambda x.\ s \\ (\lambda y.\ s)[t/x] & = & \lambda y.\ (s[t/x]) \text{ if } x \neq y \end{array}$$

However this isn't quite right. For example $(\lambda y.\, x+y)[y/x] = \lambda y.\, y+y$, which doesn't correspond to the intuitive answer.[2] The original lambda term was 'the function that adds x to its argument', so after substitution, we might expect to get 'the function that adds y to its argument'. What we actually get is 'the function that doubles its argument'. The problem is that the variable y that we have substituted is *captured* by the variable-binding operation $\lambda y. \ldots$. We should first rename the bound variable:

$$(\lambda y.\ x+y) = (\lambda w.\ x+w)$$

and only now perform a naive substitution operation:

$$(\lambda w.\ x+w)[y/x] = \lambda w.\ y+w$$

We can take two approaches to this problem. Either we can add a condition on all instances of substitution to disallow it wherever variable capture would occur, or we can modify the formal definition of substitution so that it performs the appropriate renamings automatically. We will opt for this latter approach. Here is the definition of substitution that we use:

$$\begin{array}{rcl} x[t/x] & = & t \\ y[t/x] & = & y \text{ if } x \neq y \\ c[t/x] & = & c \\ (s_1\ s_2)[t/x] & = & s_1[t/x]\ s_2[t/x] \\ (\lambda x.\ s)[t/x] & = & \lambda x.\ s \\ (\lambda y.\ s)[t/x] & = & \lambda y.\ (s[t/x]) \text{ if } x \neq y \text{ and either } x \notin FV(s) \text{ or } y \notin FV(t) \\ (\lambda y.\ s)[t/x] & = & \lambda z.\ (s[z/y][t/x]) \text{ otherwise, where } z \notin FV(s) \cup FV(t) \end{array}$$

The only difference is in the last two lines. We substitute as before in the two safe situations where either x isn't free in s, so the substitution is trivial, or where y isn't free in t, so variable capture won't occur (at this level). However where these conditions fail, we first rename y to a new variable z, chosen not to be free in either s or t, then proceed as before. For definiteness, the variable z can be chosen in some canonical way, e.g. the lexicographically first name not occurring as a free variable in either s or t.[3]

[2] We will continue to use infix syntax for standard operators; strictly we should write $+\ x\ y$ rather than $x+y$.

[3] Cognoscenti may also be worried that this definition is not in fact, *primitive* recursive,

2.3.4 Conversions

Lambda calculus is based on three 'conversions', which transform one term into another one intuitively equivalent to it. These are traditionally denoted by the Greek letters α (alpha), β (beta) and η (eta).[4] Here are the formal definitions of the operations, using annotated arrows for the conversion relations.

- Alpha conversion: $\lambda x.\, s \underset{\alpha}{\longrightarrow} \lambda y.\, s[y/x]$ provided $y \notin FV(s)$. For example, $\lambda u.\, u\ v \underset{\alpha}{\longrightarrow} \lambda w.\, w\ v$, but $\lambda u.\, u\ v \underset{\alpha}{\not\longrightarrow} \lambda v.\, v\ v$. The restriction avoids another instance of variable capture.
- Beta conversion: $(\lambda x.\, s)\ t \underset{\beta}{\longrightarrow} s[t/x]$.
- Eta conversion: $\lambda x. t\ x \underset{\eta}{\longrightarrow} t$, provided $x \notin FV(t)$. For example $\lambda u. v\ u \underset{\eta}{\longrightarrow} v$ but $\lambda u.\, u\ u \underset{\eta}{\not\longrightarrow} u$.

Of the three, β-conversion is the most important one to us, since it represents the evaluation of a function on an argument. α-conversion is a technical device to change the names of bound variables, while η-conversion is a form of *extensionality* and is therefore mainly of interest to those taking a logical, not a programming, view of lambda calculus.

2.3.5 Lambda equality

Using these conversion rules, we can define formally when two lambda terms are to be considered equal. Roughly, two terms are equal if it is possible to get from one to the other by a finite sequence of conversions (α, β or η), either forward or backward, at any depth inside the term. We can say that lambda equality is the *congruence closure* of the three reduction operations together, i.e. the smallest relation containing the three conversion operations and closed under reflexivity, symmetry, transitivity and substitutivity. Formally we can define it inductively as follows, where the horizontal lines should be read as 'if what is above the line holds, then so does what is below'.

$$\frac{s \underset{\alpha}{\longrightarrow} t \text{ or } s \underset{\beta}{\longrightarrow} t \text{ or } s \underset{\eta}{\longrightarrow} t}{s = t}$$

$$\frac{}{t = t}$$

because of the last clause. However it can easily be modified into a primitive recursive definition of multiple, parallel, substitution. This procedure is analogous to strengthening an induction hypothesis during a proof by induction. Note that by construction the pair of substitutions in the last line can be done in parallel rather than sequentially without affecting the result.

[4] These names are due to Curry. Church originally referred to α-conversion and β-conversion as 'rule of procedure I' and 'rule of procedure II' respectively.

$$\frac{s = t}{t = s}$$

$$\frac{s = t \text{ and } t = u}{s = u}$$

$$\frac{s = t}{s\ u = t\ u}$$

$$\frac{s = t}{u\ s = u\ t}$$

$$\frac{s = t}{\lambda x.\ s = \lambda x.\ t}$$

Note that the use of the ordinary equality symbol (=) here is misleading. We are actually *defining* the relation of lambda equality, and it isn't clear that it corresponds to equality of the corresponding mathematical objects in the usual sense.[5] Certainly it must be distinguished sharply from equality at the *syntactic* level. We will refer to this latter kind of equality as 'identity' and use the special symbol $\equiv$. For example $\lambda x.\ x \not\equiv \lambda y.\ y$ but $\lambda x.\ x = \lambda y.\ y$.

For many purposes, α-conversions are immaterial, and often $\equiv_\alpha$ is used instead of strict identity. This is defined like lambda equality, except that only α-conversions are allowed. For example, $(\lambda x.\ x)y \equiv_\alpha (\lambda y.\ y)y$. Many writers use this as identity on lambda terms, i.e. consider equivalence classes of terms under $\equiv_\alpha$. There are alternative formalizations of syntax where bound variables are unnamed (de Bruijn 1972), and here syntactic identity corresponds to our $\equiv_\alpha$.

2.3.6 Extensionality

We have said that η-conversion embodies a principle of *extensionality*. In general philosophical terms, two properties are said to be *extensionally* equivalent (or *coextensive*) when they are satisfied by exactly the same objects. In mathematics, we usually take an extensional view of sets, i.e. say that two sets are equal precisely if they have the same elements. Similarly, we normally say that two functions are equal precisely if they have the same domain and give the same result on all arguments in that domain.

As a consequence of η-conversion, our notion of lambda equality is extensional. Indeed, if $f\ x$ and $g\ x$ are equal for any x, then in particular $f\ y = g\ y$ where y is chosen not to be free in either f or g. Therefore by the last rule above, $\lambda y.\ f\ y = \lambda y.\ g\ y$. Now by η-converting a couple of times at depth, we see that

[5] Indeed, we haven't been very precise about what the corresponding mathematical objects *are*. But there are models of the lambda calculus where our lambda equality is interpreted as actual equality.

$f = g$. Conversely, extensionality implies that all instances of η-conversion do indeed give a valid equation, since by β-reduction, $(\lambda x.\, t\ x)\ y = t\ y$ for any y when x is not free in t. This is the import of η-conversion, and having discussed that, we will largely ignore it in favour of the more computationally significant β-conversion.

2.3.7 Lambda reduction

Lambda equality, unsurprisingly, is a symmetric relation. Though it captures the notion of equivalence of lambda terms well, it is more interesting from a computational point of view to consider an asymmetric version. We will define a 'reduction' relation $\longrightarrow$ as follows:

$$\frac{s \underset{\alpha}{\longrightarrow} t \text{ or } s \underset{\beta}{\longrightarrow} t \text{ or } s \underset{\eta}{\longrightarrow} t}{s \longrightarrow t}$$

$$\frac{}{t \longrightarrow t}$$

$$\frac{s \longrightarrow t \text{ and } t \longrightarrow u}{s \longrightarrow u}$$

$$\frac{s \longrightarrow t}{s\ u \longrightarrow t\ u}$$

$$\frac{s \longrightarrow t}{u\ s \longrightarrow u\ t}$$

$$\frac{s \longrightarrow t}{\lambda x.\, s \longrightarrow \lambda x.\, t}$$

Actually the name 'reduction' (and one also hears β-conversion called β-reduction) is a slight misnomer, since it can cause the size of a term to grow, e.g.

$$\begin{aligned}
(\lambda x.\, x\ x\ x)\ (\lambda x.\, x\ x\ x) &\longrightarrow (\lambda x.\, x\ x\ x)\ (\lambda x.\, x\ x\ x)\ (\lambda x.\, x\ x\ x) \\
&\longrightarrow (\lambda x.\, x\ x\ x)\ (\lambda x.\, x\ x\ x)\ (\lambda x.\, x\ x\ x)\ (\lambda x.\, x\ x\ x) \\
&\longrightarrow \ldots
\end{aligned}$$

However reduction does correspond to a systematic attempt to evaluate a term by repeatedly evaluating combinations $f(x)$ where f is a lambda abstraction. When no more reductions except for α conversions are possible we say that the term is in *normal form*.

2.3.8 Reduction strategies

Let us recall, in the middle of these theoretical considerations, the relevance of all this to functional programming. A functional program is an *expression* and executing it means evaluating the expression. In terms of the concepts discussed here, we are proposing to start with the relevant term and keep on applying reductions until there is nothing more to be evaluated. But how are we to choose which reduction to apply at each stage? The reduction relation is not deterministic, i.e. for some terms t there are several t_i such that $t \longrightarrow t_i$. Sometimes this can make the difference between a finite and infinite reduction sequence, i.e. between a program terminating and failing to terminate. For example, by reducing the innermost *redex* (reducible expression) in the following, we have an infinite reduction sequence:

$$\begin{array}{ll} & (\lambda x.\ y)\ ((\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)) \\ \longrightarrow & (\lambda x.\ y)\ ((\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)) \\ \longrightarrow & (\lambda x.\ y)\ ((\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)) \\ \longrightarrow & \cdots \end{array}$$

and so ad infinitum. However the alternative of reducing the outermost redex first gives:

$$(\lambda x.\ y)\ ((\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)) \longrightarrow y$$

immediately, and there are no more reductions to apply.

The situation is clarified by the following theorems, whose proofs are too long to be given here. The first one says that the situation we have noted above is true in a more general sense, i.e. that reducing the leftmost outermost redex is the best strategy for ensuring termination.

Theorem 2.2 *If $s \longrightarrow t$ with t in normal form, then the reduction sequence that arises from s by always reducing the leftmost outermost redex is guaranteed to terminate in normal form.*

Formally, we define the 'leftmost outermost' redex recursively: for a term $(\lambda x.\ s)\ t$ it is the term itself; for any other term $s\ t$ it is the leftmost outermost redex of s, and for an abstraction $\lambda x.\ s$ it is the leftmost outermost redex of s. In terms of concrete syntax, we always reduce the redex whose λ is the furthest to the left.

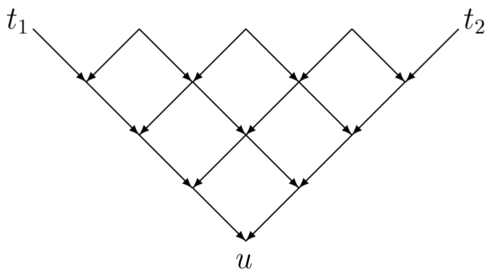
2.3.9 The Church-Rosser theorem

The next assertion, the famous Church-Rosser theorem, states that if we start from a term t and perform any two finite reduction sequences, there are always two more reduction sequences that bring the two back to the same term (though of course this might not be in normal form).

Theorem 2.3 *If $t \longrightarrow s_1$ and $t \longrightarrow s_2$, then there is a term u such that $s_1 \longrightarrow u$ and $s_2 \longrightarrow u$.*

This has at least the following important consequences:

Corollary 2.4 *If $t_1 = t_2$ then there is a term u with $t_1 \longrightarrow u$ and $t_2 \longrightarrow u$.*
Proof: *It is easy to see (by structural induction) that the equality relation $=$ is in fact the symmetric transitive closure of the reduction relation. Now we can proceed by induction over the construction of the symmetric transitive closure. However less formally minded readers will probably find the following diagram more convincing:*



We assume $t_1 = t_2$, so there is some sequence of reductions in both directions (i.e. the zigzag at the top) that connects them. Now the Church-Rosser theorem allows us to fill in the remainder of the sides in the above diagram, and hence reach the result by composing these reductions. Q.E.D.

Corollary 2.5 *If $t = t_1$ and $t = t_2$ with t_1 and t_2 in normal form, then $t_1 \equiv_\alpha t_2$, i.e. t_1 and t_2 are equal apart from α conversions.*
Proof: *By the first corollary, we have some u with $t_1 \longrightarrow u$ and $t_2 \longrightarrow u$. But since t_1 and t_2 are already in normal form, these reduction sequences to u can only consist of alpha conversions. Q.E.D.*

Hence normal forms, when they do exist, are unique up to alpha conversion. This gives us the first proof we have available that the relation of lambda equality isn't completely trivial, i.e. that there are any unequal terms. For example, since $\lambda x\ y.\ x$ and $\lambda x\ y.\ y$ are not interconvertible by alpha conversions alone, they cannot be equal.

Let us sum up the computational significance of all these assertions. In some sense, reducing the leftmost outermost redex is the best strategy, since it will work if any strategy will. This is known as *normal order reduction.* On the other hand *any* terminating reduction sequence will always give the same result, and moreover it is never too late to abandon a given strategy and start using normal order reduction. We will see later how this translates into practical terms.

2.4 Combinators

Combinators were actually developed as an independent theory by Schönfinkel (1924) before lambda notation came along. Moreover Curry rediscovered the theory soon afterwards, independently of Schönfinkel and of Church. (When he found out about Schönfinkel's work, Curry attempted to contact him, but by that time, Schönfinkel had been committed to a lunatic asylum.) We will distort the historical development by presenting the theory of combinators as an aspect of lambda calculus.

We will define a *combinator* simply to be a lambda term with no free variables. Such a term is also said to be *closed*; it has a fixed meaning independent of the values of any variables. Now we will later, in the course of functional programming, come across many useful combinators. But the cornerstone of the theory of combinators is that one can get away with just a few combinators, and express in terms of those and variables *any* term at all: the operation of lambda abstraction is unnecessary. In particular, a closed term can be expressed purely in terms of these few combinators. We start by defining:

$$
\begin{aligned}
I &= \lambda x.\ x \\
K &= \lambda x\ y.\ x \\
S &= \lambda f\ g\ x.\ (f\ x)(g\ x)
\end{aligned}
$$

We can motivate the names as follows.[6] I is the identity function. K produces constant functions:[7] when applied to an argument a it gives the function $\lambda y.\ a$. Finally S is a 'sharing' combinator, which takes two functions and an argument and shares out the argument among the functions. Now we prove the following:

Lemma 2.6 *For any lambda term t not involving lambda abstraction, there is a term u also not containing lambda abstractions, built up from S, K, I and variables, with $FV(u) = FV(t) - \{x\}$ and $u = \lambda x.\ t$, i.e. u is lambda-equal to $\lambda x.\ t$.*

Proof: *By structural induction on the term t. By hypothesis, it cannot be an abstraction, so there are just three cases to consider.*

- *If t is a variable, then there are two possibilities. If it is equal to x, then $\lambda x.\ x = I$, so we are finished. If not, say $t = y$, then $\lambda x.\ y = K\ y$.*
- *If t is a constant c, then $\lambda x.\ c = K\ c$.*
- *If t is a combination, say $s\ u$, then by the inductive hypothesis, there are lambda-free terms s' and u' with $s' = \lambda x.\ s$ and $u' = \lambda x.\ u$. Now we claim $S\ s'\ u'$ suffices. Indeed:*

[6] We are not claiming these are the historical reasons for them.

[7] Konstant — Schönfinkel was German, though in fact he originally used C.

$$\begin{aligned} S\ s'\ u'\ x &= S\ (\lambda x.\ s)\ (\lambda x.\ u)\ x \\ &= ((\lambda x.\ s)\ x)((\lambda x.\ u)\ x) \\ &= s\ u \\ &= t \end{aligned}$$

Therefore, by η-conversion, we have $S\ s'\ u' = \lambda x.\ S\ s'\ u'\ x = \lambda x.\ t$, since by the inductive hypothesis x is not free in s' or u'.

Q.E.D.

Theorem 2.7 *For any lambda term t, there is a lambda-free term t' built up from S, K, I and variables, with $FV(t') = FV(t)$ and $t' = t$.*
Proof: *By structural induction on t, using the lemma. For example, if t is $\lambda x.\ s$, we first find, by the inductive hypothesis, a lambda-free equivalent s' of s. Now the lemma can be applied to $\lambda x.\ s'$. The other cases are straightforward. Q.E.D.*

This remarkable fact can be strengthened, since I is definable in terms of S and K. Note that for any A:

$$\begin{aligned} S\ K\ A\ x &= (K\ x)(A\ x) \\ &= (\lambda y.\ x)(A\ x) \\ &= x \end{aligned}$$

So again by η-converting, we see that $I = S\ K\ A$ for any A. It is customary, for reasons that will become clear when we look at types, to use $A = K$. So $I = S\ K\ K$, and we can avoid the use of I in our combinatory expression.

Note that the proofs above are constructive, in the sense that they guide one in a definite procedure that, given a lambda term, produces a combinator equivalent. One proceeds bottom-up, and at each lambda abstraction, which by construction has a lambda-free body, applies the top-down transformations given in the lemma.

Although we have presented combinators as certain lambda terms, they can also be developed as a theory in their own right. That is, one starts with a formal syntax excluding lambda abstractions but including combinators. Instead of α, β and η conversions, one posits *conversion rules* for expressions involving combinators, e.g. $K\ x\ y \longrightarrow x$. As an independent theory, this has many analogies with lambda calculus, e.g. the Church-Rosser theorem holds for this notion of reduction too. Moreover, the ugly difficulties connected with bound variables are avoided completely. However the resulting system is, we feel, less intuitive, since combinatory expressions can get rather obscure.

Apart from their purely logical interest, combinators have a certain practical potential. As we have already hinted, and as will become much clearer in the later chapters, lambda calculus can be seen as a simple functional language, forming the core of real practical languages like ML. We might say that the theorem of combinatory completeness shows that lambda calculus can be 'compiled down' to a 'machine code' of combinators. This computing terminology is not as fanciful as it appears. Combinators have been used as an implementation technique for functional languages, and real hardware has been built to evaluate combinatory expressions.

Further reading

An encyclopedic but clear book on lambda calculus is Barendregt (1984). Another popular textbook is Hindley and Seldin (1986). Both these contain proofs of the results that we have merely asserted. A more elementary treatment focused towards computer science is Part II of Gordon (1988). Much of our presentation here and later is based on this last book.

Exercises

1. Find a normal form for $(\lambda x\ x\ x.\ x)\ a\ b\ c$.

2. Define $twice = \lambda f\ x.\ f(fx)$. What is the intuitive meaning of $twice$? Find a normal form for $twice\ twice\ twice\ f\ x$. (Remember that function application associates to the left.)

3. Find a term t such that $t \underset{\beta}{\longrightarrow} t$. Is it true to say that a term is in normal form if and only if whenever $t \longrightarrow t'$ then $t \equiv_\alpha t'$?

4. What are the circumstances under which $s[t/x][u/y] \equiv_\alpha s[u/y][t/x]$?

5. Find an equivalent in terms of the S, K and I combinators alone for $\lambda f\ x.\ f(x\ x)$.

6. Find a *single* combinator X such that all λ-terms are equal to a term built from X and variables. You may find it helpful to consider $A = \lambda p.p\ K\ S\ K$ and then think about $A\ A\ A$ and $A\ (A\ A)$.

7. Prove that any X is a fixed point combinator if and only if it is itself a fixed point of G, where $G = \lambda y\ m.\ m(y\ m)$.