Introduction to Functional Programming

John Harrison

University of Cambridge

Lecture 2

λ-calculus as a formal system

Topics covered:

- Why a formal system?
- Free and bound variables
- Substitution
- Conversion rules and lambda equality
- Reduction, evaluation strategies and the Church-Rosser theorem(s)
- Combinators

Why a formal system?

We have taken certain facts as 'obvious', e.g.

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

because they respect the intuitive meanings. Now we specify some *formal* rules for calculating with lambda expressions.

These can then be applied mechanically, without thinking about them, just as one might take:

$$x - 3 = 5 - x$$

and transform it, without considering the underlying meanings, to

$$2x = 5 + 3$$

Whitehead quotation

On the subject of symbolisms and formal rules of manipulation:

> [...] 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.

A. N. Whitehead: 'An Introduction to Mathematics', 1919.

Formal syntax of λ-terms

Lambda terms are built up inductively in the following ways:

- **Variables**, e.g. x, u and v_3.
- **Constants**, e.g. c, 0 and $false$.
- **Applications**, of the form $s\ t$ where s and t are λ-terms, e.g. $f(x)$.
- **Abstractions**, of the form $\lambda x.\ s$, e.g. $\lambda x.\ x$.

This can be described by the following BNF grammar:

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

Since the syntax is defined inductively, we can *define things by primitive recursion* and *prove things by structural induction.*

Primitive recursion: FV

As an illustration of how to define functions recursively over the syntax of λ-terms, here is a formal definition of the set of free variables in a term t, written $FV(t)$.

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

In general, we can always define the value of a function on a given term in terms of the value on the immediate subterms, as here. Intuitively, it is clear that we can always calculate the value of the function on any particular term.

Structural induction: FV is finite

As an illustration of structural induction, we will prove that for any λ-term t, the set $FV(t)$ is always finite.

- If t is a variable v, then $FV(t) = \{v\}$ by definition, and this is certainly finite.
- If t is a constant c, then $FV(t) = \emptyset$, which is trivially finite.
- If t is an application $s_1\ s_2$, then by the inductive hypothesis $FV(s_1)$ and $FV(s_2)$ are both finite. But $FV(t) = FV(s_1) \cup FV(s_2)$ and the union of two finite sets is finite.
- If t is an abstraction $\lambda x.\ s$, then by the inductive hypothesis $FV(s)$ is finite. But $FV(t) = FV(s) - \{x\}$, which must also be finite as it is no larger.

Q.E.D.

Substitution

In order to express certain rules, we need to formalize the notion of substituting a term for a variable in another term.

We write $t[s/x]$ for the result of substituting a term s for a variable x in another term t. One also sees $t[x := s]$ and even $t[x/s]$.

Ours can be remembered by thinking of multiplication of fractions:

$$x[t/x] = t$$

For example:

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

Of course we only substitute for *free* variables, so $(\lambda x.\, x)[y/x] = \lambda x.\, x$.

Naive substitution

At first sight we can define substitution by primitive recursion as follows:

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

However this isn't right. We get $(\lambda y.\ x + y)[y/x] = \lambda y.\ y + y$ — *variable capture.* We should rename first to $\lambda w.\ x + w$, and only then perform the naive substitution:

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

Renaming substitution

So we consider two cases when defining $(\lambda y.\, s)[t/x]$ for $x \neq y$.

- If either $x \notin FV(s)$ or $y \notin FV(t)$ then we can proceed as before; variable capture will not occur.
- Otherwise, we pick a new variable $z \notin FV(s) \cup FV(t)$ and form $\lambda z.\, (s[z/y][t/x])$. That is, first we rename the bound variable y to z, then proceed with the substitution as before.

For definiteness, we can define z to be the lexicographically earliest variable not occurring free in s or t.

The above definition ensures that substitution always respects the intuitive interpretation. Now we can use this operation freely.

Conversion rules

Lambda calculus is based on three fundamental '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). The most important to us is β.

- 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 $\lambda u.\ u\ v \underset{\alpha}{\not\longrightarrow} \lambda v.\ v\ v$. This 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 $\lambda u.\ u\ u \underset{\eta}{\not\longrightarrow} u$.

Lambda equality

We say that two terms s and t are equal, and write $s = t$, if there is a finite sequence of α, β or η conversions, forwards or backwards, at any depth, which connects them.

Formally we define the relation inductively (see next slide).

Note that this 'equality' is a *defined* notion and is not the same as equality at the syntactic level. We'll call this 'identity', and use the symbol $\equiv$. For example $\lambda x.\ x = \lambda y.\ y$ but $\lambda x.\ x \not\equiv \lambda y.\ y$.

The equality relation is *extensional*, i.e. if two functions f and g give equal results for all arguments, then they are themselves equal. Indeed, if f and g have this property, then choose a variable y not free in f or g. We have $f\ y = g\ y$ and therefore $\lambda y.\ f\ y = \lambda y.\ g\ y$. But now by a couple of η-conversions, we get $f = g$.

Lambda equality: definition

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

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

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

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

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

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

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

Lambda reduction

This is a 'directional' version of lambda equality.

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

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

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

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

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

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

Evaluation strategy

The idea is that reduction corresponds to a systematic attempt to evaluate expressions by applying functions to arguments. However there are often several different ways to reduce a given term. For example we have:

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

ad infinitum. However we can reduce the leftmost outermost *redex* first and get simply y, which is in normal form, i.e. no nontrivial reductions are possible.

In general, then, evaluation strategy is practically important, and we will return to this subject.

The Church Rosser theorem

This says that if $s \longrightarrow t_1$ and $s \longrightarrow t_2$, then there is a u such that $t_1 \longrightarrow u$ and $t_2 \longrightarrow u$.

By applying this repeatedly, we get the stronger form that if $t_1 = t_2$ then there is a term u with $t_1 \longrightarrow u$ and $t_2 \longrightarrow u$.

So *if* a term reduces to, or in general is equal to, a term in normal form, that normal form is unique up to alpha conversion.

Also a consequence of the theorem is that lambda equality is nontrivial, because two terms in normal form that are not alpha-equivalent are not equal. For example $\lambda x\ y.\ x \neq \lambda x\ y.\ y$.

Another theorem (also proved by Church and Rosser) is that if *any* reduction sequence terminates, then the one arrived at by systematically reducing the leftmost outermost redexes will terminate.

Combinators

It turns out that we can dispense with λ-abstractions altogether if we introduce three very simple *combinators*:

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

The names can be remembered by I = Identity, K = Konstant and S = Sharing.

Combinatory completeness: every λ-term has an equivalent, with the same free variables, written using combinators and with no lambda abstractions.

This is quite attractive, because it avoids all the complications like variable capture.

Combinatory completeness

It's enough to be able to find a combinatory equivalent for any $\lambda x.\ t$ where t is λ-free. Then we can apply the procedure repeatedly to the innermost abstraction in a bottom-up fashion.

- If t is a variable, then either $t = x$, when $\lambda x.\ x = I$, or $t \neq x$, when $\lambda x.\ y = K\ y$.
- If t is a constant, say c, then $\lambda x.\ c = K\ c$.
- If t is an abstraction, say $s\ u$, then by the inductive hypothesis, there are lambda-free terms s' and u' with $s' = \lambda x.\ s$ and $u' = \lambda x.\ u$. A simple calculation shows that $\lambda x.\ s\ u = S\ s'\ u'$.

In fact, S and K alone suffice, since $I = S\ K\ K$.