33 Lambda Calculus

The lambda-calculus was developed by Alonzo Church in the 1930s, at the same time as Turing was thinking about computability. Church was also thinking about computability, but he approached in a logical style rather than Turing’s machine-based style. In fact, Church was the first to show the existence of undecidable problems – his proof was published a couple of months earlier than Turing’s.

Lambda-calculus was the basic for the programming language LISP, and also underlies the functional languages such as ML and Haskell. It reduces computation explicitly to manipulation of primitive symbols, by means of functions constructed from only two basic operations. In this form, it is not convenient for real-life, so in practice one usually adds primitive data types such as numbers, strings, etc., but everything can be done in the basic language.

We start with the language itself.

Definition 74 The untyped lambda-calculus is a set of terms given by the following rules:

1. There is a set of variables \( x, y, \ldots \). Every variable is a term.
2. If \( x \) is a variable and \( t \) is a term, then \( (\lambda x.t) \) is a term. Such a term is called an abstraction.
3. If \( s \) and \( t \) are terms, then \( (st) \) is a term. Such a term is called an application.

In writing terms, the following conventions allow omitting parentheses:

1. Omit outermost parentheses. (E.g. \( \lambda x.t \))
2. Application associates to the left: \( stu \) means \( (st)u \).
3. The body of an abstraction extends as far to the right as possible: \( \lambda x.s\lambda y.xty \) means \( \lambda x.(s(\lambda y.(xty))) \) ◀

In traditional imperative programming languages, and in general mathematical usage, functions are defined by first giving them names, and then writing out a definition (e.g. \( f(x) = 3x + 2 \)). In lambda-calculus, abstraction allows functions to be created as first-class objects, without giving them names: \( \lambda x.t \) is function that takes an argument \( s \), say, binds \( s \) to the variable \( x \), and evaluates \( t \).

Application is application of a function to an argument, and the process of application involves binding the argument to the variable. The notion of binding, and the associated notion of \( \alpha \)-conversion, require some book-keeping.

Definition 75 If a variable \( x \) occurs inside a term \( t \), the occurrence is free in \( t \) iff \( x \) is not inside a subterm \( \lambda x.t' \) of \( t \). If \( x \) is free in \( t' \), it is bound in \( \lambda x.t' \). ◀

When doing formal proofs about lambda-calculus, one produces more formal definitions by precise inductive definitions on terms. These require some care to get right, and are not otherwise interesting, so we will omit them from this course. Details can be found in the course text, or on-line. One of the main uses of such definitions is making precise the intuitively obvious fact that there is no substantive difference between \( \lambda x.x \) and \( \lambda y.y \).

Definition 76 An \( \alpha \)-conversion of a term \( t \) is an instance of the following procedure: let \( \lambda x.t' \) be a subterm of \( t \), and let \( y \) be a variable that does not occur free in \( t' \). Replace the binding occurrence of \( x \), and every free occurrence of \( x \) in \( t' \), by \( y \).

We write \( s \xrightarrow{\alpha} t \) if \( s \) can be converted to \( t \) by a sequence of \( \alpha \)-conversions. ◀
From now on, we will treat terms as equivalent if they can be $\alpha$-converted to each other.

Computation in the lambda-calculus is done by applying abstractions to arguments, in a way that is essentially syntactic substitution. Because of the way variables are bound, one has to take some care with the substitution.

**Definition 77** Let $t = (\lambda x.t')s$ be an application. The $\beta$-reduction $u$ of the $\beta$-redex $t$ is constructed thus: first, if necessary $\alpha$-convert $(\lambda x.t')$ to $(\lambda x'.t'')$ so that every bound variable in it is distinct from every free variable of $s$. Then $u$ is the result of substituting $s$ for every free occurrence of $x'$ in $t''$.

We write $t \xrightarrow{\beta} u$. ◁

Another conversion that is sometimes used allows us to ‘wrap’ a function inside another abstraction – this can be used to obtain fine control over the order of evaluation.

**Definition 78** $\eta$-equivalence is generated by the following equation: provided that $x$ is not free in $f$, then $(\lambda x.fx) \xrightarrow{\eta} f$. ◁

When we evaluate lambda-terms by doing $\beta$-reduction, the question arises: when there are several possible $\beta$-redexes, which one do we reduce first? Or, indeed, do we reduce redexes that are hidden inside abstractions, or only those we see ‘at the top level’?

For example, consider the term

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

in which both the first and second abstractions could be functions in a redex. If we reduce the first redex, we get

$$(\lambda z.w)((\lambda x.xx)(\lambda x.xx)) \xrightarrow{\beta} w$$

whereas if we reduce the second redex, we get

$$(\lambda z.w)((\lambda x.xx)(\lambda x.xx)) \xrightarrow{\beta} (\lambda z.w)((\lambda x.xx)(\lambda x.xx))$$

and thus a never-ending sequence of reductions that make no change. The fundamental difference is whether we evaluate the argument of a function before applying the function (as in most imperative programming languages), or whether we apply the function to the unevaluated argument (as in Haskell and other ‘lazy’ languages).

**Definition 79** An evaluation strategy is a rule which, given a lambda-term, decides which $\beta$-redex to reduce next.

*Call by name* is the following strategy: give a term $t$,

- if $t$ is $(\lambda x.u)v$, $\beta$-reduce it; otherwise,
- if $t$ is $uv$ (where $u$ is not an abstraction), reduce (following the strategy) in $u$ if possible; otherwise,
- reduce, if possible, in $v$, following the strategy.

*Call by value* is the following strategy: given a term $t$,

- if $t$ is $(\lambda x.u)v$, reduce in $v$ if possible, otherwise $\beta$-reduce it; otherwise,
- if $t$ is $uv$ (where $u$ is not an abstraction), reduce in $u$ if possible.

Note that neither of these strategies ever looks inside an abstraction.
34 Numbers etc. in the lambda-calculus

This lambda-calculus is clearly very spartan – it shuffles symbols around, but apparently has nothing about numbers and other basic constructs of computing.

For practical purposes, it is convenient to use an extended language, where, for example, we add basic numerals $0, 1, 2, \ldots$ as symbols of the language, as well as arithmetic operators. We can build these into the semantics of the language by adding clauses to the definition of $\beta$-reduction, such as

$$+n_1n_2 \xrightarrow{\beta} n \text{ where } n = n_1 + n_2$$

(and if we are in call-by-name setting, we have to adjust the strategy so that these operations work in a call-by-value way, so that expressions such as $+\left(\left(+\,1\,1\right)\right)$ reduce).

Similarly, we do not (apparently) have any way to write conditional expressions in the language. We could solve this by again adding primitive operators to the language with associated $\beta$-reduction rules. For example, we might add a three-argument function $\text{if}$, with the intended meaning that $\text{if}bst$ means ‘if $b$ then $s$ else $t$’, where $b$ should reduce a value of some primitive boolean type (or we could take a C-like view, and say that it must evaluate to a number, where 0 means false and anything else means true):

$$\text{ifbst} \xrightarrow{\beta} s \text{ if } b \text{ is true}$$

$$t \text{ if } b \text{ is false}$$

and again, in a call-by-name setting, we need to force $b$ to be evaluated in call-by-value style.

However, it is not necessary to do this: it is possible to represent numbers (and booleans, and complex data structures) by coding them into the language as we have it. There are several possible ways to do this – the best known, the Church numerals, is discussed on the tutorial sheet associated with this lecture.

35 Recursion

Another aspect of computing that is not obviously included in the language we have introduced is that of recursion or iteration. How do we write a while-loop, or a recursive function? In programming language design, we usually again take the approach of extending the language with new primitives; but in this section, we shall see that we don’t actually need to.

The obvious obstacle to defining recursive functions is that we don’t have named functions, only anonymous abstractions. The trick to get round this is to use another abstraction: suppose that $t$ is a term representing a function (so, an abstraction itself). Suppose that $u$ is a term which does something with a function variable $f$. Then if we consider $(\lambda f.u)(t)$, when we evaluate $u$, the variable $f$ will, by the $\beta$-reduction rule, effectively be a name for the function $t$. This lets us write code using named functions, and in particular lets us, with a bit of effort, write functions that can call themselves.

To expand on this a bit, consider the example from the slides. In a normal recursive definition, the factorial function would be defined as

$$f(x) = \text{if } x = 0 \text{ then } 1 \text{ else } x \times f(x - 1)$$

In lambda-calculus, the function that takes $x$ and returns $x!$ would be

$$\lambda x.\text{if } x = 0 \text{ then } 1 \text{ else } x \times f(x - 1)$$
– except that we haven’t defined \( f \)! So we add another layer of abstraction, and write something that takes another function as an argument:

\[
\lambda f. \lambda x. \text{if } x = 0 \text{ then } 1 \text{ else } x \times f(x - 1).
\]

But what do we give it as an argument? \( f \) needs to be \( \lambda x. \ldots \), but that in turn uses a different \( f \) (because of \( \alpha \)-conversion, so we seem to be stuck.

Several famous computer scientists are alleged to have said ‘all problems in computer science can be solved by adding another level of indirection’ (I heard it attributed to David Wheeler), and this is no exception to that rule. The \( \text{Y} \) combinator works its magic by using a higher-order function which takes the \( \lambda f. \lambda x. \ldots \) as an argument, and succeeds in feeding it to itself in the right way.

**Definition 80** Let \( \text{Y} \) denote the lambda-term

\[
\text{Y} \equiv \lambda F. (\lambda X. F(X)(X))((\lambda X. F(X)))
\]

What is the \( F \) argument to \( \text{Y} \)? It should be a function that itself takes two arguments, the first of which (\( f \), say) is a function that takes an argument (\( x \), say), and the second of which is a suitable \( x \) to give to \( f \); as in the factorial function above. The result of applying \( \text{Y} \) to this will be a function which in turn takes a value for \( x \).

So suppose we have a function \( \lambda f. \lambda x. u \), where \( u \) mentions \( f \) and \( x \). What happens if we apply \( \text{Y} \) to this term as the first argument, and to some value \( x_0 \) for the second argument, and start \( \beta \)-reducing?

\[
\begin{align*}
\text{Y}(\lambda f. \lambda x. u) x_0 & \\
& \overset{\beta}{\rightarrow} (\lambda X. (\lambda f. \lambda x. u)(X)(X))(\lambda X. (\lambda f. \lambda x. u)(X)) x_0 \\
& \overset{\beta}{\rightarrow} (\lambda f. \lambda x. u)((\lambda X. (\lambda f. \lambda x. u)(X)(X))(\lambda X. (\lambda f. \lambda x. u)(X)))(X) x_0 \\
& \overset{\beta}{\rightarrow} 2 u[((\lambda X. (\lambda f. \lambda x. u)(X)(X))(\lambda X. (\lambda f. \lambda x. u)(X)))(X)]/f, x_0/x
\end{align*}
\]

Now we will start evaluating the function body \( u \), with \( x_0 \) as the value of \( x \). If, in the course of evaluation, we see what was originally \( f \) in the function body, it has been replaced by

\[
((\lambda X. (\lambda f. \lambda x. u)(X)(X))(\lambda X. (\lambda f. \lambda x. u)(X)))
\]

which is what we have in the second line above, so the derivation will continue like that, with the new ‘\( x_0 \)’ being whatever argument is given to \( f \) in the term \( u \).

The key to this very intricate derivation is the observation made on the slides, that

\[
\text{YG} \overset{\beta}{\rightarrow} (\lambda X. G(X)(X))(\lambda X. G(X)) \overset{\beta}{\rightarrow} G((\lambda X. G(X)(X))(\lambda X. G(X)))
\]

and the later is one reduction of \( G(\text{YG}) \).

The slides work through this for the factorial function, expressed in proper lambda style

\[
\text{if } (= 0 x) 1 (\times x (f(-x 1)))
\]

rather than the more easily readable

\[
\text{if } x = 0 \text{ then } 1 \text{ else } x \times f(x - 1)
\]

we used above. You should complete the derivation yourself to make sure you see how it works.