Lecture 10: Untyped λ-calculus

Summary.
(1) Introduction to the λ-calculus.

(2) Basic concepts in *untyped* λ-calculus: terms; α—equivalence; β—reduction as a computational dynamics.

(3) A glimpse on programming within the untyped λ-calculus.

Luís Soares Barbosa,
UNIV. MINHO *(Informatics Dep.)* & INL *(Quantum Software Engineering Group)*

Note.

This lecture initiates the last part of the course. The objective is to introduce the Curry-Howard-Lambek correspondence connecting Logic, Computation and Categories, the later providing the basic mathematical semantic structures. The triangle below will be discussed for the two computation paradigms students have met along their Physics Engineering degree: classical and quantum.

$$\begin{tikzcd}
\text{Computation} \arrow[rr, leftrightarrow] \arrow[dr, leftrightarrow] & & \text{Categories} \arrow[dl, leftrightarrow] \\
& \text{Logic} &
\end{tikzcd}$$

Overview.

The λ-calculus [1] is a theory of *functions seen as formal expressions*, and therefore somehow closer to the *intensional* view of *functions as rules* (e.g. $f(x) = \sqrt{x \sin x}$) which was predominant in the pre-20th century Mathematics. Note that in a discipline of programming this view is relevant to typical computational questions concerning the way a function is defined, often irrespectively of its actual meaning. For example, questions concerned with how much memory or time the execution of a function takes?[1]. Treating functions as expressions makes possible to nest them without any need to mention the intermediate results explicitly, as well as to take them as first class citizens, and thus easily express higher-order functions.

The λ-calculus was initially proposed by Alonzo Church, around 1930, as an idealized programming language and postulated to be able to represent any *computable* function. Even if the notion

[1] The alternative, more general view, brought to scene by the development of set theory, focus on the way arguments are mapped to outputs. Functions are regarded as graphs in this *extensional* perspective.

of a *computable* function is only given intuitively[2], the class of functions expressible in the λ-calculus coincides with that of Gödel class of *general recursive functions* as well as the one defined by Turing machines. The assertion that this class of functions, expressed in any of these formal models, captures intuitive computability is known as the Church-Turing thesis.

This lecture introduces the *untyped* version of the λ-calculus which omits any information on the *type*, i.e. domain and codomain, of a function. This provides a very flexible, although possibly unsafe setting to manipulate and reason about functions.

λ-terms.

Given a countably infinite set of variables, X, the set Λ, which provides the syntax for the λ-calculus, contains the terms built inductively according to the following grammar:

$$t, t' \ni x \mid t\,t' \mid \lambda x.t$$

where $x \in X$.

Conventions.

- Application associate to the left — e.g. $f\,x\,y$ means $(f\,x)\,y$.
- The scope of an abstraction goes as far to the right as possible.

Free variables. Variables not bound by an abstraction are *free* (they correspond to the *assumptions* within a term).

$$\begin{aligned} \mathcal{FV}(x) &= \{x\} \\ \mathcal{FV}(t\,t') &= \mathcal{FV}(t) \cup \mathcal{FV}(t') \\ \mathcal{FV}(\lambda x.t) &= \mathcal{FV}(t) \setminus \{x\} \end{aligned}$$

Variable renaming and α-equivalence.

Two terms are α-equivalent if they differ solely in the bounded variables. The relation $=_\alpha$ is defined as the smallest congruence satisfying the following rule

$$\frac{y \notin t}{\lambda x.t \;=\; \lambda y.t[x := y]}\,(\alpha)$$

[2] An informal definition of computability calls for a 'pencil-and-paper' method allowing a trained person to calculate the result of the function for any given argument, is not easy to formalize.

where

$$
\begin{aligned}
z[x := y] &= \begin{cases} y & \Leftarrow z = x \\ z & \Leftarrow \text{otherwise} \end{cases} \\
(t\,u)[x := y] &= t[x := y]\,u[x := y] \\
(\lambda z.t)[x := y] &= \begin{cases} \lambda y.t[x := y] \;\Leftarrow\; z = x \\ \lambda z.t[x := y] \;\Leftarrow\; \text{otherwise} \end{cases}
\end{aligned}
$$

and $y \notin t$ abbreviates *variable* y *not occurring free in term* t.

Every term is α-equivalent to another term in which the names of all bound variables are distinct from each other and from any free variable (the proof follows an easy inductive argument). In practice, we may always assume, without loss of generality, that bound variables can be renamed to be distinct.

Substitution of v for x in t. Substitution of a term v for a variable x in another term t, represented by $t[x := v]$ must be done carefully. Firstly, only free variables can be replaced, e.g.

$$(x(\lambda x.\lambda y.x))\,[x := v] \;=\; v(\lambda x.\lambda y.x)$$

and not $v(\lambda v.\lambda y.v)$. Additionally, free variables cannot be *captured* along the substitution. For example, let $v = \lambda z.x\,z$ and consider the following substitution,

$$\lambda x.y\,x\,[y := v] \;=\; \lambda x.v\,x \;=\; \lambda x.(\lambda z.x\,z)\,x$$

Note that variable x was free in term v and got captured along the substitutions.

Formally,

$$
\begin{aligned}
z[x := v] &= \begin{cases} v \;\Leftarrow\; z = x \\ z \;\Leftarrow\; \text{otherwise} \end{cases} \\
(t\,u)[x := v] &= (t[x := v])\,(u[x := v]) \\
(\lambda x.t)[x := v] &= \lambda x.t \\
(\lambda y.t)[x := v] &= (\lambda y.t[x := v]) \quad \Leftarrow\; x \neq y \text{ and } y \notin \mathcal{FV}(v) \\
(\lambda y.t)[x := v] &= (\lambda z.t[y := z][x := v]) \quad \Leftarrow\; x \neq y \text{ and } y \in \mathcal{FV}(v) \text{ and } z \text{ fresh}
\end{aligned}
$$

Exercise 1

The composition of a function with itself $(f \cdot f)$ is written in the λ-calculus as

$$\lambda x.f\,f\,x$$

Encode (higher-order) functions to map f to $f \cdot f$, and the pair of functions f and g to $f \cdot g$.

$\boxed{\textbf{Exercise } 2}$

Evaluate the expression

$$((\lambda f.\lambda x.f(f\,x)))\,(\lambda y.y^2))(2)$$

Note that the expression above is not a pure λ-expression (why?)

$\boxed{\textbf{Exercise } 3}$

Which of the following pairs of terms are α-equivalent?

$$\{(\lambda x.xz, \lambda y.yz), (\lambda x.\lambda y.xy, \lambda y.\lambda x.yx), (\lambda x.xy, \lambda x.xz)\}$$

$\boxed{\textbf{Exercise } 4}$

Show that $=_\alpha$ is an equivalence relation over Λ. Note that, strictly speaking, λ-terms are the classes of equivalence in the quotient

$$\Lambda/{=_\alpha} = \{[t]_\alpha \mid t \in \Lambda\} = \{\{u \in \Lambda \mid t =_\alpha u\} \mid t \in \Lambda\}$$

In some textbooks elements of Λ are called λ-*pre-terms*.

$\boxed{\textbf{Exercise } 5}$

Compute

1. $(\lambda x.x\,y)[x := \lambda z.z]$
2. $(\lambda x.x\,y)[y := \lambda z.z]$

λ **dynamics.**

β**-reduction.** There is a *computational* dynamics captured by the λ-calculus: that of *functional* application. Formally, β-reduction is the smallest relation on λ-terms such that

$$\underbrace{(\lambda x.t)\,u}_{\beta-\mathit{redex}} \longrightarrow_\beta \underbrace{t[x := u]}_{\beta-\mathit{contractum}}$$

and is closed under the following rules: if $t \longrightarrow_\beta t'$, then, for all $x \in X$ and λ-term v,

$$\begin{aligned} t\,u &\longrightarrow_\beta t'\,u \\ u\,t &\longrightarrow_\beta u\,t' \\ \lambda x.t &\longrightarrow_\beta \lambda x.t' \end{aligned}$$

A term t is in a *normal* form if there is no term u such that $t \longrightarrow_\beta u$.

The Church-Rosser Theorem (1936).

If a term t has two derivations, e.g. $t \longrightarrow_\beta^* v$ and $t \longrightarrow_\beta^* v'$, there exists a term u such that $v \longrightarrow_\beta^* u$ and $v' \longrightarrow_\beta^* u$.

Extensionality: η**-equivalence.** The terms x and $\lambda y.x\,y$, being normal forms for $\longrightarrow_\beta$, are not β-equivalent. In general the same applies to t and $\lambda y.t\,y$, for an arbitrary term t. However, both terms 'exhibit the same behaviour' and, thus, from the point of view of *extensionality* should be equivalent. This can be captured by η-reductions: the smallest congruence satisfying the following rule:

$$\frac{}{\lambda y.t\,y \longrightarrow_\eta t}\ (\eta)$$

whenever $y \notin \mathcal{FV}(t)$. Note that $\beta\eta$-reduction is the union of both relations. Similarly, one defines $=_{\beta\eta}$ and normal form for $\beta\eta$-reduction.

$\boxed{\textbf{Exercise } 6}$

Compute β-reductions of the following terms

1. $\lambda x.y\,((\lambda z.z\,z)\,(\lambda w.w))$
2. $(\lambda x.x\,x)\,\lambda z.z$
3. $(\lambda z.z)\,\lambda y.y$

Exercise 7

Define β-equivalence, $=_\beta$, as the transitive, reflexive, symmetric closure of $\longrightarrow_\beta$. Show that

$$(\lambda x.x)\, y\, z \;=_\beta\; y\, ((\lambda x.x)\, z)$$

Exercise 8

Define $\mathbf{I} = \lambda x.x$ and $\mathbf{K} = \lambda y.\lambda x.y$. Show that the term $\mathbf{K}(\mathbf{II})$ contains more than one redex and can, thus, have more than one β-reduction. Show also that both derivations converge in the same term.

Exercise 9

Show that the term $\Omega = \omega\omega$, where $\omega = \lambda x.xx$, has an infinite β-reduction sequence.

Expressability.

As discussed above, β-reduction expresses (classical, functional) computation. In the following half-worked exercises we discuss how to represent arithmetic, Booleans, conditionals and recursive definitions within the λ-calculus. As mentioned above, the λ-calculus constitutes an alternative formulation of the theory of recursive functions, which by the Church-Turing thesis, captures the notion of an *effectively computable* procedure, just as a Turing machine. We will not formalised that discussion here, as irrelevant for the objective of this course. The interested reader is referred to e.g. [2].

Exercise 10

Numerals. The number n can be represented in the λ-calculus by the following term, known as the *Church numeral* n,

$$c_n \;=\; \lambda s.\lambda z.s^n\, z$$

Write the Church numerals corresponding to numbers 0 to 3. Show that

$$\mathsf{succ}\, c_n \;=\; c_{n+1}$$

given the following encoding of the successor function:

$$\mathsf{succ} \;=\; \lambda n.\lambda f.\lambda x.f\,(n\,f\,x).$$

$\boxed{\textbf{Exercise } 11}$

Arithmetic. Consider now the following encoding of addition

$$\mathit{add} \;=\; \lambda x.\lambda y.\lambda s.\lambda z.x\, s\, (y\, s\, z)$$

Verify that

$$\begin{aligned}
\mathit{add}\, c_n c_m &= (\lambda x.\lambda y.\lambda s.\lambda z.xs(ysz))\, c_n c_m \\
&=_\beta \lambda s.\lambda z.c_n s(c_m sz) \\
&=_\beta \lambda s.\lambda z.c_n s(s^m z) \\
&=_\beta \lambda s.\lambda z.s^n(s^m z) \\
&= \lambda s.\lambda z.s^{n+m} z \\
&= c_{n+m}
\end{aligned}$$

Study the following encodings of multiplication and exponentiation:

$$\mathit{mult} \;=\; \lambda x.\lambda y.\lambda s.x(ys) \quad \text{and} \quad \mathit{exp} \;=\; \lambda x.\lambda y.yx$$

$\boxed{\textbf{Exercise } 12}$

Booleans. The Boolean values true and false can be encoded as

$$\mathit{true} \;=\; \lambda x.\lambda y.x \quad \text{and} \quad \mathit{false} \;=\; \lambda x.\lambda y.y$$

show that the term $\mathit{and} = \lambda xy.x\, y\, \mathit{false}$ encodes conjunction.

$\boxed{\textbf{Exercise } 13}$

Conditionals. Conditionals can be represented by the term

$$b \to t; u \;=\; b\, t\, u$$

for $b \in \{\mathit{true}, \mathit{false}\}$. Actually,

$$\begin{aligned}
\mathit{true} \to t; u &= \mathit{true}\, t\, u \\
&= (\lambda x.\lambda y.x) t\, u \\
&=_\beta (\lambda y.t)\, u \\
&=_\beta t
\end{aligned}$$

Compute $\mathit{false} \to t; u$.

Exercise 14

Pairing. Consider the following encoding of pairs

$$\begin{aligned}\langle t, u\rangle &= \lambda x.xtu \\ \pi_1 &= \lambda x.\lambda y.x \\ \pi_2 &= \lambda x.\lambda y.y\end{aligned}$$

Show that $\langle t, u\rangle\, \pi_1 \;=\; t$

Exercise 15

Recursion. A fixed point of a function f is a value x such that $x = f(x)$. Their relevance comes from this: they are solutions to equations. Similarly, we may say that a term u is a fixed point of a term t in the λ-calculus if

$$u \;=_\beta\; t\,u$$

Differently to what happens in arithmetic, in the untyped λ-calculus, every term t has a fixed point. which means that one can always solve equations as above in the calculus.

The following λ-term encodes a *fixed point operator*:

$$Y \;=\; \lambda f.(\lambda x.f\,(x\,x))\,\lambda x.f\,(x\,x)$$

Actually,

$$\begin{aligned}Y\,t &= (\lambda f.(\lambda x.f\,(x\,x))\,\lambda x.f\,(x\,x))\,t \\ &=_\beta (\lambda x.t\,(x\,x))\,\lambda x.t\,(x\,x)) \\ &=_\beta t\,((\lambda x.t\,(x\,x))\,\lambda x.t\,(x\,x)) \\ &=_\beta t\,((\lambda f.(\lambda x.f\,(x\,x))\,\lambda x.f\,(x\,x))\,t) \\ &= t\,(Y\,t)\end{aligned}$$

As a corollary note that, for a given term v, there is a term t such that

$$t \;=_\beta\; v[f := t]$$

Show that such is the case by making

$$t \;=\; Y\,(\lambda f.v)$$

Exercise 16

Let C be a term encoding a condition, i.e. $\mathsf{cond}\,c_n =_\beta \mathsf{true}$ or $\mathsf{cond}\,c_n =_\beta \mathsf{false}$, for all $n \in \mathbb{N}$. Define

$$H \;=\; \lambda f.\lambda x.\,((\mathsf{cond}\,x) \to x; f(\mathsf{succ}\,x))$$

The term $\mathsf{R} = \mathsf{Y}\,\mathsf{H}$ corresponds to the computation of the smallest number greater than the given one that satisfies condition cond. Study the following derivation:

$$\begin{aligned}
\mathsf{R}\,c_4 &= (\mathsf{Y}\,\mathsf{H})\,c_4 \\
&=_\beta \mathsf{H}\,(\mathsf{Y}\,\mathsf{H})\,c_4 \\
&= (\lambda f.\lambda x.\,(\mathsf{cond}\,x \to x; f\,(\mathsf{succ}\,x)))\,(\mathsf{Y}\,\mathsf{H})\,c_4 \\
&=_\beta (\mathsf{cond}\,c_4 \to c_4; (\mathsf{Y}\,\mathsf{H})\,(\mathsf{succ}\,c_4)) \\
&= \mathsf{cond}\,c_4 \to c_4; \mathsf{R}\,(\mathsf{succ}\,c_4)
\end{aligned}$$

References

[1] H. Barendregt. *The Lambda-Calculus: Its Syntax and Semantics*. Elsevier Science Publishers B. V. (North-Holland), 1980.

[2] J.R. Hindley and J.P. Seldin. *Lambda-calculus and Combinators: an Introduction*. Cambridge University Press, 2008.