A.  λ-calculus

The λ-calculus is a core calculus that was primarily designed to investigate function definition, function application and recursion. As a core calculus, its design emphasizes simplicity and minimality, however, it is capable of expressing any algorithm thus is equivalent to Turing machine in terms of its expressive power. Since its introduction in the 1930s by Church and Kleene, it has emerged as an important tool in computability and recursion theory. It is defined by the following three ways of forming terms.

\[ e ::= \]

- \( x \) “Variable, where \( x \in \mathcal{V} \), the countably infinite set of variables.”
- \( \lambda x.e \) “Abstraction, the λ-calculus equivalent of \( x \mapsto e \).”
- \( e_0 e_1 \) “Application, the function \( e_0 \) applied to argument \( e_1 \).”

The use of terms instead of expressions here (and throughout this course) will clearly separate executable expressions from non-executable expressions. The use of term will be limited to all executable expressions and will exclude all other non-executable expressions e.g. type expressions.

Note that the λ-abstraction is the only binding construct in this calculus in the sense that in \( \lambda x.e \), \( \lambda \) binds values of \( x \) appearing in \( e \) to the value of the argument \( x \).

Also, note that function application \( e_0 e_1 \) is the only computational construct in this calculus.

A.1  α-conversion

In λ-calculus two terms that differ only in the name of variables used are considered equivalent. For example, \( \lambda x.x \) and \( \lambda y.y \) are considered to be equivalent. A conversion of lambda term \( \lambda x.x \) to an equivalent lambda term with different variable names e.g. \( \lambda y.y \) is called α-conversion.

A.2  β-reduction

Now we will study how to run λ-calculus programs. To that end, two concepts are important: redex i.e. a reducible expression, and β-reduction. A term of the form \( (\lambda x.t) t' \) is reducible in the sense that it can be reduced by substituting \( t' \) in place of \( x \) in \( t \) as demonstrated below:

\[ (\lambda x.t)t' \rightarrow^\beta [x \mapsto t']t \]

Here, \( [x \mapsto t']t \) means “the term obtained by replacing all free occurrences of \( x \) in \( t \) by \( t' \)”. For example, for \( (\lambda x.x z)y \), there is only one free occurrence of \( x \) in \( x z \), thus after replacing the resulting term is \( y z \). The operation of rewriting a redex according to the rule above is called β-reduction. We will represent β-reduction by \( \rightarrow^\beta \) in this course. There are several strategies for evaluation of λ-calculus programs. We will discuss them in greater detail in the next lecture.

Recap: higher-order functions.  As discussed previously, functions that take other functions as argument or return other functions are called higher-order functions. An example of higher-order function is \( \lambda x.\lambda y.x \). When applied to another term, it returns another “abstraction”. Consider the evaluation of \( (\lambda x.\lambda y.x)a \) below:

\[
(\lambda x.\lambda y.x)a \\
= ([x \mapsto a])(\lambda y.x)
\]
In the initial few lecture notes, to make the β-reduction explicit, we will use underlining to identify the terms that are being reduced and also show the intermediate state. In the reduction above, second line shows the intermediate state where the term $x$ in $\lambda y.x$ is marked to be replaced by the term $a$.

Recap: currying. The transformation of multiple argument functions into higher-order functions. Here is an example:

$$\lambda x.y.x y := (\lambda x.(\lambda y.x y))$$

The general form for a n-argument function is $$(\lambda x_0.(\lambda x_1.(\ldots \lambda x_n(body)\ldots)))$$, where the $i$th term consumes one argument and returns a term capable of consuming $i - 1$ arguments.

B. Idioms in Lambda Calculus

Now we will study some of the common programming idioms in lambda calculus that serve to demonstrate its expressive power.

Church Booleans. A common built-in data type is Boolean. This data type can be encoded in the lambda calculus. Generally the Boolean data type is incorporated into a programming language design by including representations for two constants that stand for true and false. For convenience, conditionals such as if $b$ then $e$ else $e'$ and operators like logical conjunction and disjunctions are also defined.

The constants true and false can be defined as:

$$\text{true} ::= \lambda t.\lambda f.t$$
$$\text{false} ::= \lambda t.\lambda f.f$$

The terms true and false that we just defined can be used for testing the truth of a boolean value, thus for all practical purposes these are adequate representation of the boolean constants. Using true and false, we can represent the conditional like if $l$ then $m$ else $n$ as follows:

$$\text{test} ::= \lambda l.\lambda m.\lambda n. l m n$$

Let us see how β-reduction of test works for (test fls v w). For β-reduction of (test true v w) see [1, page 59].

$$\text{test fls v w}$
$$= (\lambda l.\lambda m.\lambda n.l m n) \text{ fls v w}$
$$= (\lambda l.\lambda m.\lambda n.l m n) (\lambda t.\lambda f.f) v w$$
$$\rightarrow \beta (\lambda m.\lambda n.\lambda l.\lambda f.f) m n) v w$$
$$= (\lambda m.\lambda n.\lambda l.\lambda f.f) m n) w$$
$$\rightarrow \beta (\lambda f.f) v w$$
$$= (\lambda t.\lambda f.f) v w$$
$$\rightarrow \beta (\lambda f.f) v w$$
$$= (\lambda t.\lambda f.f) v w$$
$$\rightarrow \beta (\lambda f.f) v w$$
$$= (\lambda t.\lambda f.f) v w$$
$$\rightarrow \beta (\lambda f.f) v w$$
$$= (\lambda f.f) v w$$
$$\rightarrow \beta (\lambda f.f) v w$$
$$= [f \mapsto w] f$$
$$\rightarrow \beta w$$
Let us define logical conjunction \( \text{and}(b\;c) \), disjunction \( \text{or}(b\;c) \) and negation \( \text{not}(b) \) in terms of boolean constants defined above:

\[
\text{or} := \lambda b.\lambda c. \text{tru}\;c \\
\text{and} := \lambda b.\lambda c. b\;c\;\text{fls} \\
\text{not} := \lambda b. b\;\text{fls}\;\text{tru}
\]

The definitions of these are fairly straightforward.

**Exercise B.1** Define the logical operator xor in the lambda calculus. You can use the definitions of Church booleans, conditional, and other operators.

**Pairs.** One can encode pair of values as terms in \( \lambda \)-calculus as follows:

\[
\text{pair} := \lambda f.\lambda s.\lambda b. b\;f\;s \\
\text{fst} := \lambda p. \text{tru} \\
\text{snd} := \lambda p. \text{fls}
\]

Let us now assume a pair \( \text{pair}\;v_0\;v_1 \). The \( \beta \)-reduction for \( (\text{fst}\;\text{pair}\;v_0\;v_1) \) is shown by Pierce [1, page 60], below we show the \( \beta \)-reduction for \( (\text{snd}\;\text{pair}\;v_0\;v_1) \).

\[
(\text{snd}\;\text{pair}\;v_0\;v_1) \\
= (\lambda p. \text{fls})(\lambda f.\lambda s.\lambda b. b\;f\;s)(v_0\;v_1) \\
= (\lambda p. \text{fls})((\lambda f.\lambda s.\lambda b. b\;f\;s)) (v_0\;v_1) \\
\rightarrow_\beta (\lambda p. \text{fls})(\lambda s.\lambda b. b\;v_0\;v_1) \\
= (\lambda p. \text{fls})(\lambda s.\lambda b. b\;v_0\;v_1) \\
\rightarrow_\beta (\lambda p. \text{fls})(\lambda s.\lambda b. b\;v_0\;v_1) \\
= (\lambda p. \text{fls})(\lambda s.\lambda b. b\;v_0\;v_1) \\
\rightarrow_\beta (\lambda p. \text{fls})(\lambda s.\lambda b. b\;v_0\;v_1) \\
= \text{fls}(b\;v_0\;v_1) \\
\rightarrow_\beta \text{fls}(v_0\;v_1) \\
= \lambda t.\lambda f. f\;v_0\;v_1 \\
= \lambda t.\lambda f. f\;v_0\;v_1 \\
\rightarrow_\beta \lambda f. v_1 \\
= \lambda f. v_1 \\
\rightarrow_\beta v_1
\]

**Exercise B.2** Define a term \( \text{flip} \), which given a pair returns another pair with swapped terms. For example, the result of evaluating \( \text{flip}\;\text{pair}\;v_0\;v_1 \) is \( \text{pair}\;v_1\;v_0 \). Show an example \( \beta \)-reduction for \( \text{flip}\;\text{pair}\;v_0\;v_1 \).

**Church Numerals.** So far we have only talked about representation of functions and values in lambda calculus. However, real programs use numbers and arithmetic operators among other data types. This can be easily defined in lambda calculus.

\[
0 := \lambda s.\lambda z. z
\]
Each number $n$ in this series is defined as a combinator that takes two arguments $s$ and $z$ and applies $s$, $n$ times, to $z$.

**Successor function for Church Numerals.** The key idea in the representation of church numerals was to apply the function $s$, $n$ times, to represent number $n$. The successor function thus must be a higher-order function that returns a function that applies $s$, $n + 1$ times.

\[
\text{Succ} ::= \lambda n. \lambda s. \lambda z. s \, (n \, s \, z)
\]

Here, $n$ is the combinator defined above. So $(n \, s \, z)$ is the application of $s$ to $z$, $n$ times, thus $s \, (n \, s \, z)$ is the application of $s$ to $z$, $n + 1$ times.

**Addition function for Church Numerals.** The addition function for church numerals can be similarly defined as:

\[
\text{Plus} ::= \lambda m. \lambda n. \lambda s. \lambda z. m \, s \, (n \, s \, z)
\]

This function adds two church numerals $m$ and $n$. First, it applies $s$, $n$ times to $z$. It then applies $s$, $m$ times to the result.

**Multiplication function for Church Numerals.** The definition of multiplication function for church numerals is also similar:

\[
\text{Mult} ::= \lambda m. \lambda n. \lambda s. \lambda z. m \, (n \, s) \, z
\]

The key idea in the definition of this function is the use of $(n \, s)$, which creates a combinator that applies $s$, $n$ times. This combinator is then applied to $z$, $m$ times. This is equivalent to applying $s$ on $z$, $n \times m$ times.

**Exercise B.3** Define a term `isZero` for checking whether a Church numeral is zero.

**Exercise B.4** Define a term `pred` for computing the predecessor of a Church numeral.

**Exercise B.5** Using `pred` define a term `sub` for subtraction.

**Exercise B.6** Define a term `equal` to check the equality of two Church numerals.

**Exercise B.7** Read the subsection on conversion from Church boolean/numeral to real boolean/numeral in Pierce [1, pp. 63–65].
C. Lecture Summary

Important concepts covered in today’s lecture were:

- Three rules for forming terms in a λ-calculus.
- Notion of α-conversion, redex, and β-reduction
- Higher-order functions and currying

References