Abstract
This lecture introduces the lambda calculus (\(\lambda\)-calculus). It presents grammars for the abstract and concrete syntax of \(\lambda\)-calculus, and explains the notion of computation in the \(\lambda\)-calculus in terms of string rewriting and tree rewriting.

1 Introduction

Lambda-calculus (\(\lambda\)-calculus) is a model of computation invented by Alonzo Church in the 1930’s. It is a stripped-down functional programming language—basically as stripped down as could possibly be: it has only three kinds of constructs. Nevertheless, \(\lambda\)-calculus is a Turing-complete language.

\(\lambda\)-calculus is built on two concepts:

- **abstraction**: the forming of functional expressions (“\(\lambda\)-terms”)
- **application**: the use of a functional expression by applying it to an argument

There is one \(\lambda\)-calculus construct for each; the third kind of \(\lambda\)-calculus construct consists of variables.

Programs are functional expressions; the data domain consists of functional expressions. In other words, every program can be operated on as a piece of data, and every piece of data can be interpreted as a program.

The \(\lambda\)-calculus has no other kinds of data (e.g., numbers, lists, strings); however, we can performing encodings in which certain subsets of \(\lambda\)-terms can be identified as representing, e.g., the natural numbers, and other \(\lambda\)-terms represent the familiar functions for operating on natural numbers (e.g., plus, times). In other words, we encode an “abstract data type” by designing \(\lambda\)-terms that operate in a way that mimics the data and functions of the abstract data type.

2 Syntax

2.1 String View of \(\lambda\)-Calculus

**Definition 2.1 (\(\lambda\)-terms)** Let \(V\) be a countably-infinite set of variables. The set of \(\lambda\)-terms are defined inductively as follows:

1. every variable \(v \in V\) is a lambda-term
2. if \(M\) is a \(\lambda\) term, then so is the abstraction \((\lambda x.M)\)
3. if \(M\) and \(N\) are \(\lambda\)-terms, then so is the application \((MN)\)

Alternatively, we can express the concrete syntax of \(\lambda\)-calculus by means of the following grammar:

\[
\text{exp ::= var} \\
\quad | (\lambda \text{var.exp}) \\
\quad | (\text{exp exp})
\]

In this grammar, the left and right parenthesis symbols are part of the subject language (i.e., they are not meta-symbols of the grammar-defining formalism).
Example 2.2

(\lambda x. (\lambda y. (xy)))
(\lambda x. ((\lambda y. x)y))
(\lambda x. (x (\lambda y. (y (\lambda z. z))))))
(((\lambda x. x)(\lambda y. y))(\lambda z. z))

\hfill \Box

Precedence Rules. Notation: We use small letters (such as x, y, z) for variables, and capital letters (such as M, N, P, and Q) as meta-variables standing for typical \(\lambda\)-terms.

We omit parentheses according to the following two precedence rules:

1. Application is left associative. For instance, MNPQ stands for (((MN)P)Q). In contrast, if you want the term \(M(N(PQ))\) you can only drop the outermost pair of parentheses: \(M(N(PQ))\).

2. Application has higher precedence than abstraction. For instance, \(\lambda x. yz\) is \((\lambda x. (yz))\), not \(((\lambda x. y)(z))\).

Example 2.3 Below on the left are some examples of \(\lambda\)-calculus terms in which some parentheses have been dropped. Their equivalent fully-parenthesized versions are shown on the right.

\[
\begin{align*}
\lambda x. \lambda y. xy &\equiv (\lambda x. (\lambda y. (xy))) \\
\lambda x. (\lambda y. x)y &\equiv (\lambda x. ((\lambda y. x)y)) \\
\lambda x. \lambda y. x \lambda z. z &\equiv (\lambda x. (\lambda y. (x (\lambda z. z)))) \\
(\lambda x. x)(\lambda y. y)(\lambda z. z) &\equiv (((\lambda x. x)(\lambda y. y))(\lambda z. z))
\end{align*}
\]

\hfill \Box

Given a \(\lambda\)-term in which parentheses may have been omitted, an algorithm for fully parenthesizing the \(\lambda\)-term is to repeatedly perform the following steps:

1. Find the rightmost dot ("\.)
2. Work through the terms to the right of the dot; insert parentheses to create a left-associative list
3. Move out to the \(\lambda\) that corresponds to the dot, and place a left parenthesis to the left of the \(\lambda\) and a right parenthesis to the right of the body processed in item 2

Example 2.4

\[
\begin{align*}
\lambda x. \lambda y. \lambda x. xy \lambda z. z &\to \lambda x. \lambda y. \lambda x. xy (\lambda z. z) \quad \text{items 1, 2, and 3} \\
&\to \lambda x. \lambda y. \lambda x. ((xy)(\lambda z. z)) \quad \text{items 1 and 2} \\
&\to \lambda x. \lambda y. \lambda x. ((xy)(\lambda z. z)) \quad \text{item 3} \\
&\to \lambda x. (\lambda y. ((xy)(\lambda x. (\lambda z. z)))) \quad \text{items 1, 2, and 3} \\
&\to (\lambda x. (\lambda y. ((xy)(\lambda z. z)))) \quad \text{items 1, 2, and 3}
\end{align*}
\]

\hfill \Box

The abstract syntax of \(\lambda\)-calculus is defined by the following grammar:

\[
\begin{align*}
term &::= \text{Var}(\text{var}) \\
&\quad | \text{Lambda}(\text{var term}) \\
&\quad | \text{App}(\text{term term})
\end{align*}
\]

This can be considered to be a grammar that generates a language of trees. (Technically, it is called a regular tree grammar.) In this grammar, \text{Var}, \text{Lambda}, and \text{App} are the operators (or
alphabet symbols) from which trees are constructed. (As a shorthand, in diagrams of abstract-syntactic trees, we often use the symbol “λ” in place of the operator “Lambda”.) The left and right parenthesis symbols are meta-symbols of the grammar-defining formalism; i.e., they are part of the meta-language, not the subject language.

2.2 Tree View of λ-Calculus
Recall that the abstract syntax gives us a tree view of λ-terms. That is, we can convert \(((M N) P) Q\) from string form to the following tree:

```
  App
 /   \
App   Q
 /     \
App     P
       /   \
M      N
```

The parentheses to insert (when converting back to fully-parenthesized string form) are implied by the tree structure:

```
  App
 /   \
(App  Q )
 /     \
(App  P )
       /   \  
(M   N )
```

When applying rewrite rules, it’s usually less mistake-prone to process a λ-term as follows:

```
string-representation → tree-representation
                      → normal-form in tree-representation
                      → normal-form in string-representation.
```

3 Semantics
The intended semantics is that
- An abstraction represents a 1-argument function
- An application represents the application of a term \(M\) to input “data” \(N\)

Note that there is not a separate notion of “data” in λ-calculus. That is, all data items are themselves functional expressions: the λ-calculus consists of functional expressions that operate on functional expressions. In particular, we can look at the following λ-term:

\[(λx.M)N\]

and interpret it as the application of the function \(λx.M\), which has formal parameter \(x\) and function body \(M\), to actual parameter \(N\).
3.1 $\beta$-Reduction Rule of $\lambda$-Calculus

The $\beta$-reduction rule of $\lambda$-calculus allows us to resolve applications. Applications are resolved ("reduced") by replacing the formals that occur in the function body by copies of the actual parameter. (Thus, $\beta$-reduction is similar to the familiar programming-language notion of call-by name.)

**Definition 3.1 (Imprecise version of the $\beta$-reduction rule)**

$$(\lambda x. M)N \rightarrow_{\beta} M'$$
where $M'$ is $M$ with all occurrences of $x$ in $M$ replaced by $N$.

In a tree view this converts

![Tree View of Reduction](image)

**Definition 3.2** In $(\lambda x. M)N \rightarrow M'$, an occurrence of the left-hand side pattern (i.e., $(\lambda x. M)N$) is called a redex; after the reduction, the corresponding occurrence of the right-hand side pattern (i.e., $M'$) is called the contractum.

**Example 3.3** To give a concrete example, consider the $\beta$-reduction

$$(\lambda x(\lambda y.x))(\lambda z.z) \rightarrow \lambda y.\lambda z.z$$

In tree form, the reduction takes

![Tree View of Example Reduction](image)
4 \(\lambda\)-Calculus Syntax Redux: De Bruijn Representation

De Bruijn introduced an alternative representation for \(\lambda\)-terms in which variables are replaced by indexes (natural numbers). There is a convention that the index represents the number of levels of enclosing \(\lambda\)s that need to be traversed along the path to the root to find the \(\lambda\) with which the index (variable) is associated. The abstract syntax is changed to

\[
\text{term ::= } \text{Var}'(\text{nat}) \\
| \Lambda \text{lambda}'(\text{term}) \\
| \text{App}(\text{term} \text{ term})
\]

A tree generated by this grammar is a \(\lambda\)-calculus term in De Bruijn representation. The grammar generates trees of the following form, where the \(\lambda\) that a variable is attached to is determined by climbing up the tree towards the root, counting occurrences of \(\lambda\)s along the way.

**Example 4.1** The following figure shows (part) of a \(\lambda\)-term in De Bruijn representation, and in particular, shows how a sub-term that represents a variable occurrence is associated with an enclosing occurrence of a \(\lambda\) (i.e., Lambda) operator.

As shown by the dashed arrow, the leaf marked with the index 2 corresponds to a variable that is bound to the top \(\lambda\), because that is two levels up (where counts start from 1). \(\square\)