% Introduction to functional programming

Learning Outcomes

By the end of this class, you should be able to

- Comprehend, evaluate lambda terms
- Differentiate different evaluation strategies
- Apply lambda calculus to implement simple algorithms

Lambda Calculus

Lambda Expression

The valid syntax of lambda expression is described as the following EBNF

Where

- denotes a variable,
- denotes a lambda abstraction.
  - Within a lambda abstraction, is the bound variable (c.f. formal argument of the function) and is the body.
- denotes a function application.

Note that given a lambda term, there might be multiple ways of parsing (interpreting) it. For instance, Given , we could interpret it as either

1. , or
2.

Exercise 1

Consider the lambda term which can be parsed in multiple ways. By putting paratheses (...), show that this lambda term can be parsed in at least 2 different ways.

Free variables

Exercise 2

Assuming function application has a higher precedence level than lambda abstraction, i.e. is always parsed as , function application is left associative, i.e. is always parsed as , compute the free variables from the following lambda terms.

-
-
-

Beta Reduction

where denotes one evaluation step.
refers to a substitution. denotes the application of the substitution to , Informally speaking it means we replace every occurance of the formal argument in with .

Substitution and Alpha Renaming

In case

is not satified, we need to rename the lambda bound variables that are clashing.

Exercise 3

Apply the substitution to the following lambda terms, high-light the variables in name clashing or free variables being mis-captured if there exists; otherwise, derive the result.

-
-
-
-

Evaluation strategies

1. Inner-most, leftmost - Applicative Order Reduction
2. Outer-most, leftmost - Normal Order Reduction

Exercise 4

Apply AOR and NOR to evaluate the following lambda term.

-
-
-
-

Lambda Calculus Extended

and the evaluation rules

and free variable extended

and substitution extended

$$
\begin{array}{rcll}
\lbrack t_1 / x \rbrack x & = & t_1 \
\lbrack t_1 / x \rbrack y & = & y & {\tt if}\ x \neq y \
\lbrack t_1 / x \rbrack (t_2\ t_3) & = & \lbrack t_1 / x \rbrack t_2\

\lbrack t_1 / x \rbrack t_3 & \
\lbrack t_1 / x \rbrack \lambda y.t_2 & = & \lambda y. \lbrack t_1 / x \rbrack t_2 & {\tt if}\ y\neq x\ {\tt and}\ y \not \in fv(t_1) \
\lbrack t_1 / x \rbrack let\ y = t_2\ in\ t_3 & = & let\ y = \lbrack t_1 / x \rbrack t_2\ in\ \lbrack t_1 / x \rbrack t_3 & {\tt if}\ y\neq x\ {\tt and}\ y \not \in fv(t_1) \
\lbrack t_1 / x \rbrack if\ t_2\ then\ t_3\ else\ t_4 & = & if\ \lbrack t_1 / x \rbrack t_2\ then\ \lbrack t_1 / x \rbrack t_3\ else\ \lbrack t_1 / x \rbrack t_4 \
\lbrack t_1 / x \rbrack \mu f.t_2 & = & \mu f.\lbrack t_1 / x \rbrack t_2 & {\tt if}\ f\neq x\ {\tt and}\ f \not \in fv(t_1) \

\end{array}
$$

Exercise 5

Evaluate the following lambda term

Exercise 6

The above set of evaluation rules does not force actual arguments of function application to being evaluated first before being applying to the function body. This is known as the strict evaluation. What changes is required if we make to enforce strict evaluation

Optional Exercise - Church Encoding

Recall that Y-combinator is defined as

Exercise 7

Show that for any function , we have .

Let’s encode natural numbers. The main idea is to define

succ (AKA incr) operation

It can be defined as

The idea is that we apply one more to the input number. The input number can be accessed by applying to and .

Exercise 8

Test by evaluating to show that it equals to .

pred (AKA decr) operation

Exercise 9

Test by evaluating .

Recall that in Church encoding and are defined as

The function is rather simple.

Exercise 10

Test with and

Factorial

Recall the function implementation,

where

and

Exercise 11

Try evaluating .