

## Contents

### 1 Announcements

Your first homework is due today, so please e-mail them to the TA.

### 2 Introduction

Today we will talk about *lambda calculus*. Lambda calculus is tiny calculus that is Turing complete. It is important because it enables us both to programs and also study about the properties of such programs using mathematics.

### 3 Abstract Syntax

Let  $V$  be a countable set  $V$  of variables. We define the abstract syntax for lambda calculus as follows.

**Definition 1 (Inductive Definition)**

$\mathcal{T}$  is the least set of the terms that satisfy the following.

1. if  $x \in V$  then  $x \in \mathcal{T}$
2. if  $t_1 \in \mathcal{T}$  and  $t_2 \in \mathcal{T}$  then  $t_1 t_2 \in \mathcal{T}$
3. if  $x \in V$  and  $t \in \mathcal{T}$  then  $\lambda x.t \in \mathcal{T}$
4.  $\mathcal{T}$  is the “least” set verifying the above properties

Each term in  $\mathcal{T}$  is called a *lambda term*. Some examples are  $x$ ,  $\lambda x.x$ ,  $\lambda x.x y$ . The term  $\lambda x.x$  is also known as (*lambda*) *abstraction*, and the term  $t_1 t_2$  is known as *application*.

Note that this is the definition of the *abstract syntax*. That is, it defines the set of properly parsed terms (*i.e.*, abstract syntax trees). It does not tell us how to read a lambda term. For example, we can parse  $\lambda x.x y$  as  $(\lambda x.x y)$  or  $(\lambda x.x)y$ . Similarly, we can parse  $t_1 t_2 t_3$  as  $(t_1 t_2)t_3$  or  $t_1(t_2 t_3)$ .

We will use parenthesis to aid in parsing (to disambiguate the syntax). To minimize parenthesis, we will have the following conventions:

1. Application associates to the left.

2. The body of a lambda terms extends as far as right as possible.

With this convention  $\lambda x.x y$  is parsed as  $\lambda x.(x y)$  and  $t_1 t_2 t_3$  is parsed as  $(t_1 t_2) t_3$ .

We can define the same syntax based on inference rules:

**Definition 2 (Inference Rules)**

Given a countable set of variables  $V$ , the set of lambda terms is defined as follows.

1.  $\frac{x \in V}{x \in \mathcal{T}}$
2.  $\frac{t_1 \in \mathcal{T} \quad t_2 \in \mathcal{T}}{t_1 t_2 \in \mathcal{T}}$
3.  $\frac{x \in V \quad t \in \mathcal{T}}{\lambda x.t \in \mathcal{T}}$

The most succinct definition is the BNF style.

**Definition 3 (BNF Style)**

Assuming that  $x$  ranges over a countable set of variables, the set of lambda terms  $t$  is defined as follows.

$$t ::= x \mid t_1 t_2 \mid \lambda x.t$$

So we have defined the syntax for lambda terms but what do they mean? An intuitive way of thinking of  $\lambda x.t$  is as a function that takes  $x$  and computes the result in its body  $t$ .

To develop this intuition a bit further let's abuse our notation a bit. Suppose that our syntax allow us to write natural numbers and add them. For example, we may have terms like this  $1, 2, + 1 2$ .

Now, what does the following lambda terms do?

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

## 4 Bound and Free Variables

A lambda abstraction denotes a function. For example, the abstraction  $\lambda x.+ x 1$  denotes a function that takes a parameter,  $x$ , and returns the value  $x + 1$ . The parameter  $x$  is called the *formal parameter* and we say that  $\lambda$  *binds* it. In a lambda abstraction, the formal parameter is followed by a “.” and the *body* of the of the function. A lambda abstraction always consists of the 4 parts, the  $\lambda$ , the formal parameter, the “.” and the body.

Consider the lambda abstraction,  $\lambda x. + x y$ . In order to evaluate the function, for a particular input parameter  $x$ , we need to know the value of  $y$ . The variable  $y$  in this case is free and  $x$  is bound;  $\lambda$  binds  $x$ .

An occurrence of a variable is *bound* if there is an enclosing lambda abstraction that binds the variable, and is *free* otherwise.

#### Example 4

Some example lambda abstractions and their bound and free variables.

1.  $\lambda x.x$  is the identity function. Here  $x$  is a bound variable.
2.  $\lambda x.y$  is the constant “y” function. Here  $y$  is a free variable.
3.  $\lambda x.x y$  has  $x$  as a bound variable and  $y$  as a free variable.
4.  $\lambda x.(\lambda x.x y)$  has both variables bound.
5.  $+ x \lambda x. + x 1$ . In this term the first occurrence of  $x$  is free and the second is bound.

We define the set of *free variables* of a term,  $t$ , as the set of variables that occur free in the term and denote it with  $FV(t)$ . For example,  $FV(+ x \lambda x. + x 1) = \{x\}$ . Formally, we define the set of free variables of a term as follows.

#### Definition 5 (Free Variables)

The set of free variables of a term  $t$  is  $FV(t) \subseteq V$  is defined recursively as follows

1. If  $x \in V$  then  $FV(x) = \{x\}$
2. If  $t_1, t_2 \in \mathcal{T}$  then  $FV(t_1 t_2) = FV(t_1) \cup FV(t_2)$
3. If  $t \in \mathcal{T}$  and  $x \in V$  then  $FV(\lambda x.t) = FV(t) \setminus \{x\}$

Now that we have a definition of the set of free variables in a term, we can distinguish between two terms such as  $\lambda x.x$  and  $\lambda x. + xy$ .

#### Definition 6 (Closed and Open Terms)

A term  $t$  is closed if  $FV(t) = \emptyset$ . Otherwise  $t$  is open.

## 5 Substitution

A lambda abstraction denotes a function. How do we evaluate a function or a lambda abstraction at a particular value? For example, the term  $(\lambda x. + 1 x)2$  denotes the application of  $(\lambda x. + 1 x)$  to the parameter 2. To evaluate such a term we would like to replace the occurrences of the formal parameter  $x$  with the value 2. For example, for  $(\lambda x. + 1 x) 2$  evaluates to  $(+ 1 2)$ . This is an example substitution, where we substitute the value 2 for the variable  $x$  in  $(\lambda x. + 1 x)$ . Formally, we define substitution as follows.

**Definition 7**

The substitution of a term,  $t'$  for a variable  $x \in V$  in a term  $t$ , denoted by  $[t'/x]t$ , is an instance of  $t$  where all the free occurrences  $x$  is replaced by the term  $t'$ .

Throughout this course, we will use the notation  $[t'/x]t$  to denote a substitution of  $t'$  for  $x$  in  $t$ . Different notations are preferred by different textbooks or authors. Two other commonly used notations are  $t[t'/x]$  and  $[x := t']t$ . Your book uses the notation  $[x \rightarrow t']t$ .

**Example 8**

Some example substitutions.

1.  $[\lambda y.y/x](x\ x) = (\lambda y.y)\ (\lambda y.y)$
2.  $[\lambda y.y/x](\lambda x.x) = \lambda x.x$  ( $x$  is not free.)
3.  $[y/x](\lambda z.x) = \lambda z.y$

A substitution, as we defined it, can yield an “incorrect” result in certain cases. For example, consider the lambda abstraction  $\lambda y.x$ , the constant function, and the substitution  $(\lambda y.x)[y/x]$ , which is equal to  $\lambda y.y$ , the identity function. Thus, the substitution would naively change the constant function into the identity function. Here, the problem is that the variable  $x$ , a free variable, is substituted with  $y$ , which then becomes bound by the lambda abstraction. In this case, we say that  $y$  is *captured* in the substitution.

We do not want to transform free variables into bound variables during the substitution process. We need to redefine substitution to avoid capture. What can we do? Consider the following situation: given a program which has a function that takes a variable  $x$  and has a variable  $y$  in its body, we can change the name of  $y$  to  $z$  without any modification of the results, but we cannot change the name of  $y$  to  $x$  without adversely affecting the results. The next definition avoids possible captures.

**Definition 9 (Capture Avoiding substitution)**

$$[t/x]y = \begin{cases} t & \text{if } y = x \\ y & \text{if } y \neq x \end{cases}$$

$$[t/x](t_1\ t_2) = [t/x]t_1\ [t/x]t_2.$$

$$[t'/x](\lambda y.t) = \begin{cases} \lambda y.t & \text{if } x = y \text{ (} y \text{ is bound)} \\ \lambda y.[t'/x]t & \text{if } x \neq y \text{ and } y \notin FV(t') \end{cases}$$

**Question:** What does the substitution  $[\lambda y.x y y/z](\lambda x.x z)$  yield?

**Solution:** This substitution is undefined.

**Question:** How about the substitution  $[\lambda y.x y y/z](\lambda y.y z)$

**Solution:**  $\lambda y.y (\lambda y.x y y)$

According to this definition a substitution that causes capture is undefined. But something is just not right, because the lambda abstractions  $\lambda x.x z$  and  $\lambda y.y z$  differ only in the name of their bound variables. Lambda abstractions denote functions, thus these two lambda abstractions are the same; it should not matter what the bound variables are named.

## 6 Alpha Conversion and Alpha Equivalence

One way to avoid capture is to rename the bound variables in a lambda abstraction. The goal is to ensure that no bound variable has the same name as a free variable in the term being substituted.

This process of renaming the bound variables is called  $\alpha$ -conversion or  $\alpha$ -variation. Two terms that are reducible to each other by  $\alpha$ -conversions are *alpha equivalent*. We denote  $\alpha$ -equivalent terms  $t_1$  and  $t_2$  as  $t_1 =_\alpha t_2$ .

1.  $\lambda x.x =_\alpha \lambda y.y$ .
2.  $\lambda x.\lambda y.\lambda z.x y z =_\alpha \lambda z.\lambda y.\lambda x.z y x$ .
3.  $\lambda x.t =_\alpha \lambda y.[y/x]t$  if  $y \notin FV(t)$ .

**Exercise:** Show that  $=_\alpha$  is an equivalence relation.

Alpha conversion gives us a way to define substitution without worrying about capture.

**Definition 10 (Substitution with Explicit Alpha Conversion)**

$$[t/x]y = \begin{cases} t & \text{if } y = x \\ y & \text{if } y \neq x \end{cases}$$

$$[t/x](t_1 t_2) = [t/x]t_1 [t/x]t_2.$$

$$[t'/x](\lambda y.t) = \begin{cases} \lambda y.t & \text{if } x = y \\ \lambda z.[t'/x][z/y]t & \text{if } x \neq y \wedge z \notin FV(t) \cup FV(t') \end{cases}$$

The idea of this definition is to rename formal variable of the lambda abstraction so that it does not occur freely in  $t'$ . This ensures that no free variables of  $t'$  are captured. This is not enough, however, because we also have to make sure that we do not capture the free variables of  $t$  itself.

Note that, in this definition, substitution is a relation not a function. That is the result of a substitution is a set of terms where the renaming take different forms. More precisely, the variable  $z$  can take many values (the names of difference variables). All such terms are  $\alpha$ -equivalent.

Now that we made precise the idea of alpha conversion, we can now forget about it. From now on, we will work modulo  $\alpha$ -equivalence. That is, we will not distinguish between two terms that are  $\alpha$ -equivalent. We can now use our old definition, with implicit alpha conversion applied as required.

**Definition 11 (Substitution with Implicit Alpha Conversion)**

$$[t/x]y = \begin{cases} t & \text{if } y = x \\ y & \text{if } y \neq x \end{cases}$$

$$[t/x](t_1 t_2) = [t/x]t_1 [t/x]t_2.$$

$$[t'/x](\lambda y.t) = \begin{cases} \lambda y.t & \text{if } x = y \text{ (} y \text{ is bound)} \\ \lambda y.[t'/x]t & \text{if } x \neq y \wedge y \notin FV(t') \end{cases}$$

## 7 $\beta$ -reduction

In Section ??, we briefly discussed how to evaluate a function with a given parameter by substitution. We can give an operational semantics for lambda terms based on substitution. The idea is to take each application and reduce it to another term by applying substitution—this is called  $\beta$  reduction, and is denoted  $\rightarrow_\beta$ .

**Definition 12 (Single-step  $\beta$ -reduction)**

$$(\lambda x.t_1) t_2 \rightarrow_\beta [t_2/x]t_1$$

$$\frac{t_1 \rightarrow_\beta t'_1}{t_1 t_2 \rightarrow_\beta t'_1 t_2}$$

$$\frac{t_2 \rightarrow_\beta t'_2}{t_1 t_2 \rightarrow_\beta t_1 t'_2}$$

$$\frac{t \rightarrow_\beta t'}{\lambda x.t \rightarrow_\beta \lambda x.t'}$$

We define multi-step beta reduction, denoted  $\rightarrow_\beta^*$ , as 0 or more applications of single-step beta reduction rules. The use of  $*$  to denote reductions suggests their use in Kleene Algebra of Automata Theory.

**Definition 13 (Multi-step  $\beta$ -reduction)**

$$\begin{array}{c} t \rightarrow_{\beta}^* t \\ \\ \frac{t \rightarrow_{\beta} t'}{t \rightarrow_{\beta}^* t'} \\ \\ \frac{t_1 \rightarrow_{\beta}^* t_2 \quad t_2 \rightarrow_{\beta}^* t_3}{t_1 \rightarrow_{\beta}^* t_3} \end{array}$$

Two terms that are  $\beta$ -reducible to each other are called  $\beta$ -equivalent, denoted by  $=_{\beta}$ .

**Definition 14 ( $\beta$ -equivalence)**

$$\begin{array}{c} t =_{\beta} t \\ \\ \frac{t \rightarrow_{\beta} t'}{t =_{\beta} t'} \\ \\ \frac{t_1 =_{\beta} t_2 \quad t_2 =_{\beta} t_3}{t_1 =_{\beta} t_3} \\ \\ \frac{t =_{\beta} t'}{t' =_{\beta} t} \end{array}$$

## 8 Homework Exercise

A term  $t$  is in normal form if there is no  $t'$  such that  $t \rightarrow_{\beta} t'$ . We say that a term  $t$  is *normalizable* if there is some  $t'$  such that  $t \rightarrow_{\beta}^* t'$  and  $t'$  is in normal form. Answer the following questions and prove your answer (if your answer is yes, an example suffices).

1. Are there any normalizable terms?
2. Are there any non-normalizable terms?