LAMBDA CALCULUS

EDWARD O'CALLAGHAN

Contents

1. Prelude

TODO: Fix notation here...

- Here we shall use the symbol $\equiv$ to denote a *synonym*. For example, $x \equiv y$ reads "x is a synonym of y" and so x is synonymous with y.
- Here we shall use the notation $[y/x]exp$ to indicate the replacement of every occurrence of x with y in the expression *exp* to the right. For example, $(\lambda x.x)y = [y/x]x = y$.
- Here we shall fix the first order logical conjugations and disjunctions respectively as, $\wedge$ and $\vee$, and negation as $\neg$. True and False are denoted, **T** and **F**, respectively.

2. Introduction

The $\lambda - calculus$ formalism was introduced in the 1930s by Akonzo Church to provide a framework for the concept of effective computability. In this course we build up the rudiments of $\lambda - calculus$. We study the $\lambda - calculus$ mathematical formalism of computable functions. The properties of $\lambda - calculus$ have provided both the theoretical and rigorous foundations to functional programming.

In actual fact the whole of classical computation can be understood though the $\lambda - calculus$ formalism of the classical Turning machine. This is known as the Church-Turing thesis that sates the follow:

Conjecture 2.1 (Church-Turing thesis)**.** A function is computable if and only if it is computable by a Turing machine.

Hence, if some algorithm exists to carry out some calculation then so too can a Turing machine compute the result and hence a recursively definable function and by a λ-function.

In this way $\lambda - calculus$ can be thought of as the *smallest universal programming language*. To justify this claim we remark that $\lambda - calculus$ only consists of of *single* transformation rule called **variable substitution** and a single function definition scheme. We also make note that the transformation rule does not refer to the underlaying hardware implementation and so is independent of the actual machine implementation and so is a pure software formalisation.

The key concept that motivates this is central to mathematics, and more generally to language in general, is that of *representation*. Mathematical computation is essentially the act of symbolic re-representation based on self consistent pattern matching. The $\lambda - calculus$ is a natural formalisation to this concept and so this provides useful motivating terms to think in to familiarise yourself with the $\lambda - calculus$.

3. Definition

The $\lambda - calculus$ is defined by the concept of an "expression" defined in the following way:

Definition 3.1 (Expression)**.** An *expression* in $\lambda - calculus$ is defined recursively as:

$$
\begin{aligned}
< expression > &\doteq < name > \mid < function > \mid < application >\\
< function > &\doteq \lambda < name > . < expression >\\
< application > &\doteq < expression >< expression >
\end{aligned}
$$

where "name" (or "variable") is some arbitrary identifier.

A trivial example of a λ-expression is given here.

Example 3.2 (Identity function)**.** The following λ-expression defines the identity function:

$$\lambda x.x$$

Arguments to functions have no relevance on the behaviour of the function and so serve only as place holders to specify the substitution rule when the function is evaluated. So we can write various synonyms for the above identity function as,

$$(\lambda x.x) \equiv (\lambda y.y) \equiv (\lambda t.t)$$

and so forth.

Notice that an expression $\mathbf{E}$ is usually enclosed in parenthesis for clarity, that is $(\mathbf{E})$. We adopt the convention that function application *associates* from the left, that is we evaluate expressions in the following way:

$$\mathbf{E}_1\mathbf{E}_2\mathbf{E}_3 \ldots \mathbf{E}_n = (\ldots((\mathbf{E}_1\mathbf{E}_2)\mathbf{E}_3)\ldots\mathbf{E}_n).$$

Definition 3.3. Their are only two *keywords* used in the language, λ and the *dot.*

- The *name* after λ the identifier of the *argument* of a function.
- The *expression* after the dot is called the "body", or *definition* of a function.

The application of a function to an expression is made clear by example:

Example 3.4.

$$
\begin{aligned}
(\lambda x.x)y &= [y/x]x\\
&= y.
\end{aligned}
$$

That is, we substitute x by y in the expression to the right, which is simply x and so we have y.

4. Variables

In $\lambda - calculus$ all names are local to the definitions. In the function $\lambda x.x$ we say that x is *bound* since its occurrence in the body of the definition is preceded by λx. A name that is not preceded by a λ is said to be a *free variable.*

4.1. Free.

Formally we define a *free* variable in the following way:

Definition 4.1 (Free Variable)**.** We say that a variable $<name>$ is *free* in an expression if one of the following cases holds:

- $<name>$ is free in $<name>$.
- $<name>$ is free in $\lambda <name'> . <exp>$ if the identifier $<name>\neq<name'>$ and $<name>$ is free in $<exp>$.
- $<name>$ is free in $\mathbf{E}_1\mathbf{E}_2$ if $<name>$ is free in $\mathbf{E}_1$ or free in $\mathbf{E}_2$.

Example 4.2. Consider the expression,

$$(\lambda x.x)(\lambda y.yx).$$

Then the x in the body of the first expression from the left is bound to the first λ and the y in the body of the second expression is bound to the second λ and the x is said to be *free.*

Remark. Variables with the same name in different λ-expressions are independent. That is, x is independent in the second expression from the x in the first expression of $(\lambda x.x)(\lambda y.yx)$.

4.2. Bound.

Formally we define a *bound* variable in the following way:

Definition 4.3. We say that a variable $<name>$ is *bound* in an expression if one of the following cases hold:

- $<name>$ is bound in $\lambda <name'> . <exp>$ if the identifier $<name>=<name'>$ or if $<name>$ is bound in $<exp>$.
- $<name>$ is bound in $\mathbf{E}_1\mathbf{E}_2$ if $<name>$ is bound in $\mathbf{E}_1$ or if it is bound in $\mathbf{E}_2$.