Beta Reduction in Lambda Calculus

Magesh Kumar

January 2017

Lambda Calculus

Lambda(λ) calculus was developed by American mathematician Alonzo Church in 1930s. It is a turning complete language (i.e) any computation that can be represented by a single taped turing machine can be expressed by lambda calculus. Lambda calculus is the theoretical basis for many of the modern functional programming languages like OCaml, ML, Haskell and Lisp. It is also commonly used as the formal language of discourse in the field of programming language research. The calculus is known for its simplicity. The language consists of three simple concepts:

- Variables/Names - can be numbers or letters
- Lambda abstraction - denoted by the greek letter λ
- Application - the application of lambda asbtraction on an input term

The syntax of lambda calculus is given by:

$$\begin{aligned} \langle expression \rangle &:= \langle name \rangle | \langle function \rangle | \langle application \rangle \\ \langle function \rangle &:= \lambda \langle name \rangle . \langle expression \rangle \\ \langle application \rangle &:= \langle expression \rangle . \langle expression \rangle \end{aligned}$$

The following are all valid lambda calculus expressions:

$$\begin{aligned} \text{id} &= \lambda x.x && \text{(Identity function)} \\ \text{fst} &= \lambda x.\lambda y.x && \text{(Select first element)} \\ \text{snd} &= \lambda x.\lambda y.y && \text{(Select second element)} \\ \text{apply} &= \lambda f.\lambda x.fx && \text{(Function Application)} \end{aligned}$$

There are three different operations that can be performed on a lambda calculus expression:

- Alpha(α) Conversion
- Beta(β) Reduction
- Eta(η) Conversion

The main focus of this article will be on Beta Reduction.

Beta Reduction

The term 'Reduction', can be thought of as one step of computation (i.e) it transforms a complex expression into a simpler expression. Reduction can be applied multiple times until an expression cannot be reduced anymore. The expressions that cannot be reduced anymore are often the values of that programming language. These non-reduceable expressions are said to be in normal form.

In lambda calculus, reduction on a term containing lambda abstraction on the left is called Beta Reduction. Informally it is the application of lambda abstraction on to its arguments and is more formally defined as:

$$(\lambda x.e_1)e_2 \mapsto [e_2/x]e_1$$

The term $[e_2/x]e_1$ means substitute e_2 for every occurance of x in e_1. Some examples of beta reduction:

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

Beta Reduction is used to define the Operational Semantics of lambda calculus. It is given by three simple rules:

$$(\lambda x.e_1)e_2 \mapsto [e_2/x]e_1 \tag{1}$$

$$\begin{array}{l} e_1 \mapsto e_1' \\ \text{----} \\ e_1e_2 \mapsto e_1'e_2 \end{array} \tag{2}$$

$$\begin{array}{l} e_2 \mapsto e_2' \\ \text{----} \\ e_1e_2 \mapsto e_1e_2' \end{array} \tag{3}$$

One interesting thing to note about the semantics is that the rules are non-deterministic. They do not provide the order to apply these rule on an expression.

Capture Avoiding Substitution

In lambda calculus, the name of a variable in an expression does not play a significant role. You can replace the name of variable in an expression and still retain the meaning of the expression, this type of conversion is called alpha(α)-conversion For example, the following expressions in lambda calculus are equivalent in meaning (alpha-equivalance)

$$\lambda x.x \equiv \lambda y.y \equiv \lambda z.z \tag{1}$$
$$\lambda x.z \equiv \lambda y.z \equiv \lambda z.y \tag{2}$$

However it is possible to capture a free variable after performing an operation. A free variable is considered captured when it is placed under a lambda abstraction which binds it. This changes the meaning of the variable in a lambda expression. For example, in the expression $\lambda y.xy$ if we blindly replace the variable y with x. This captures the variable x which was initially a free variable in the expression:

$$\lambda y.xy \not\equiv \lambda x.xx$$

Variable capture can also occur when performing beta reduction. For example, if we blindly perform beta reduction in the expression $(\lambda x.(\lambda y.xy))y$, we get

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

which does not indicate the correct computation of the lambda expression because now the free variable y is captured/bound by the lambda abstraction after the substituition. One way to avoid variable capture during β-reduction is to replace the bound variable by a new variable name before performing

the reduction. For example:

$$\begin{aligned}(\lambda x.(\lambda y.xy))y & \\ &\mapsto (\lambda x.[t/y](\lambda y.xy))y \\ &\mapsto (\lambda x.(\lambda t.xt))y \\ &\mapsto [y/x]\lambda t.xt \\ &\mapsto \lambda t.yt\end{aligned}$$

which gives a completely different but correct result. You can follow this general recursive rule to perform such capture avoiding substituition:

$$\begin{aligned}[e/v]v &= v \\ [e/v]w &= w && v \neq w \\ [e/v](e_1e_2) &= [e/v]e_1[e/v]e_2 \\ [e/v](\lambda u.e_1) &= \lambda w.[e/v]([w/u]e_1)([w/u]e_1) && w \notin \{v\} \cup FV(\lambda u.e_1) \cup FV(e)\end{aligned}$$

where $FV(e)$ is the set of free variables in the expression e

Chruch Rosser Theorm

A lambda expression can have multiple reduxes (a term that can be reduced using beta reduction). Since the semantics of lambda calculus is non-deterministic (i.e) it does not give us the order to perform these reduction, there can be multiple paths taken while performing beta reduction. For example:

$$
\begin{aligned}
&(\lambda x.xx)(\underline{(\lambda xy.yx)y}) \\
&\qquad \mapsto \underline{(\lambda x.xx)(\lambda z.zy)} \\
&\qquad \mapsto \underline{(\lambda z.zy)(\lambda z.zy)} \\
&\qquad \mapsto \underline{(\lambda z.zy)y} \\
&\qquad \mapsto yy \\
&\underline{(\lambda x.xx)((\lambda xy.yx)y)} \\
&\qquad \mapsto \underline{(\lambda xy.yx)y)}(\lambda xy.yx)y) \\
&\qquad \mapsto (\lambda z.zy)\underline{(\lambda xy.yx)y)} \\
&\qquad \mapsto \underline{(\lambda z.zy)(\lambda z.zy)} \\
&\qquad \mapsto \underline{(\lambda z.zy)y} \\
&\qquad \mapsto yy
\end{aligned}
$$

We see that the final evaluation result remains the same regardless of the path taken during beta reduction. Alonzo Church and Barkley Rosser proved in 1936 that choosing the order of redux does not change the result after the final termination. This proof is commonly known as Church Rosser Theorm. The theorm is stated as follows:

> If $e \overset{*}{\mapsto} e_1$ and $e \overset{*}{\mapsto} e_2$ by arbitary sequence of reduction then there exists e_3 such that $e_1 \overset{*}{\mapsto} e_3$ and $e_2 \overset{*}{\mapsto} e_3$

An important corollary of the Church Rosser Theorm is that any lambda expression has at most one normal form.

Reduction Strategies

Since a lambda expression can have multiple reduxes, we need a strategy to choose the next redux to reduce. The number of reductions needed to arrive at the normal form depends on the startegy used. Languages based on lambda calculus employ different strategies to choose the redux. The most common strategies are:

- Applicative Order
- Normal Order

Applicative Order

In applicative order reduction, the leftmost of the innermost redux is always choosen. For example:

$$\begin{aligned}(\lambda x.xx)(\underline{(\lambda xy.yx)y}) & \\ & \mapsto \underline{(\lambda x.xx)(\lambda z.zy)} \\ & \mapsto \underline{(\lambda z.zy)(\lambda z.zy)} \\ & \mapsto \underline{(\lambda z.zy)y} \\ & \mapsto yy\end{aligned}$$

This type of reduction corresponds to "call by value" parameter passing mechanism. The advantage of using this strategy is that the arguments are evaluated exactly once. However, if we choose this strategy, it has the possiblility to end up in an expression that does not terminate. For example:

$$\begin{aligned}(\lambda x.xy)(\underline{(\lambda x.xx)(\lambda x.xx)}) & \\ & \mapsto (\lambda x.xy)(\underline{(\lambda x.xx)(\lambda x.xx)}) \\ & \mapsto (\lambda x.xy)(\underline{(\lambda x.xx)(\lambda x.xx)}) \\ & \mapsto \cdots\end{aligned}$$

Normal Order

In normal order reduction, the leftmost redex is always choosen. For example:

$$\begin{aligned}\underline{(\lambda x.xx)((\lambda xy.yx)y)} & \\ & \mapsto \underline{((\lambda xy.yx)y)}((\lambda xy.yx)y) \\ & \mapsto \underline{(\lambda z.zy)((\lambda xy.yx)y)} \\ & \mapsto \underline{((\lambda xy.yx)y)}y \\ & \mapsto \underline{(\lambda z.zy)y} \\ & \mapsto yy\end{aligned}$$

This type of reduction corresponds to "call by name" parameter passing mechanism. If a normal form exists for a lambda expression then normal

order reduction will find it. For the previous non-terminating example when we use normal order reduction strategy instead of applicative order, we get:

$$\begin{aligned}\underline{(\lambda x.xy)((\lambda x.xx)(\lambda x.xx))} & \\ & \mapsto \underline{(\lambda x.xx)y} \\ & \mapsto yy\end{aligned}$$

One disadvantage of using normal order reduction is that the reduxes on the arguments maybe copied. For example:

$$\begin{aligned}\underline{(\lambda x.xx)((\lambda y.y)t)} & \\ & \mapsto ((\lambda y.y)t)((\lambda y.y)t)\end{aligned}$$

This causes duplicate computation of the lambda term, in this case $((\lambda y.y)t)$ needs to be computed twice to get to the normal form. Thus requiring more steps to arrive at the normal form.

Conclusion

β-reduction plays an important role in understanding the semantics of lambda calculus. It along with the reduction strategies defines the basis for evaluating a lambda calculus expression.

References

http://www.cs.yale.edu/homes/hudak/CS201S08/lambda.pdf
http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf
https://en.wikipedia.org/wiki/Lambda_calculus
http://web.engr.oregonstate.edu/~walkiner/teaching/cs581-fa16/
http://prl.ccs.neu.edu/blog/2016/11/02/beta-reduction-part-1/