The Lambda Calculus

6CCS3COM Computational Models

Josh Murphy

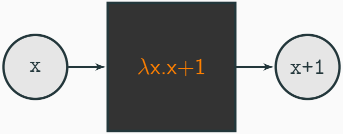
Contents

The Lambda Calculus and Turing Machines

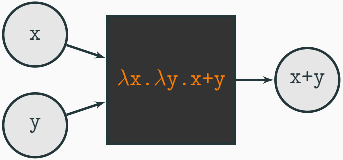
- The Turing Machine is an **imperative** computational model.
- The Lambda Calculus is a **functional** computational model.
- They are computationally equivalent models.
  - **Church-Turing thesis:** a function is λ-computable if and only if it is Turing-computable.

Functions in the Lambda Calculus

- Functions are **black-boxes**.
    - They do not have an internal representation or implementation.
    - They can be seen as pure operations.
- *e.g.* successor function



- *e.g.* addition



Applying Lambda functions

- A function can be applied to values.
  - *e.g.* $(\lambda x.x + 1)2 = 2 + 1 = 3$
    - The value 2 is substituted into x.
  - *e.g.* $(\lambda x.\lambda y.x + y)5\ 6 = 5{+}6 = 11$
    - The value 5 is substituted into x.
    - The value 6 is substituted into y.

Lambda Calculus notation

- Terms in the Lambda calculus are built from three components:
  1. **Variables**
     - We assume variables are taken from an infinite set $\{x, y, z, ...\}$
  2. **Abstractions**
     - If x is a variable and M is a term, then $(\lambda x.M)$ is a term.
  3. **Applications**
     - If M and N are terms then (M N) is a term.

Lambda Calculus notation

- Omit brackets where there is no ambiguity
- Application associates to the *left*
  - Instead of writing $((MN)P)$ we will simply write MNP
- Abstraction associates to the *right*
  - Instead of writing $\lambda x.(\lambda y.M)$ we simply write $\lambda x.\lambda y.M$, or just $\lambda xy.M$
- We assume the scope of λ is as wide as possible.
  - $\lambda x.xy = \lambda x.(xy)$
  - $\lambda x.xy \neq (\lambda x.x)y$

Example terms in the Lambda Calculus

- x
- $\lambda x.x$
- $\lambda xy.x$
- $\lambda xy.y$
- $\lambda xy.xy$
- $\lambda x.xx$
- $\lambda x.y$
- $\lambda x.yx$
- $\lambda xyz.xz(yz)$

Free and Bound variables

- In the context of a given term, a variable can be either **free** or **bound**.
- You can think about free and bound variables by relating them to the concept of program scope.
  - Bound variables are similar to local variables (their value is defined by the given context)
  - Free variables are similar to global variables (their value is defined outside of the given context)

$$\lambda x.xy$$

Formal definition of free variables

- We define the set of free variables of a term M, $FV(M)$, as a recursive function.

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x.M) &= FV(M) - \{x\} \\ FV(MN) &= FV(M) \cup FV(N) \end{aligned}$$

- Terms without free variables are **combinators** or **closed terms**.

Formal definition of bound variables

- We define the set of bound variables of a term M, $BV(M)$, as a recursive function.

$$
\begin{aligned}
BV(x) &= \emptyset = \{\} \\
BV(\lambda x.M) &= \{x\} \cup BV(M) \\
BV(MN) &= BV(M) \cup BV(N)
\end{aligned}
$$

- **Exercises**
    - In the example terms of the Lambda Calculus, which variables are free, and which are bound?
    - What are the free and bound variables of the term $x\lambda x.x$

Variable renaming and α-equivalence

- Variables can be **renamed**.
- We can rename a bound variable in a term without changing its meaning, as long as we rename it **consistently**.
- For example, $\lambda x.yx$ and $\lambda z.yz$ are computationally identical.
- $\lambda x.yx$ and $\lambda x.zx$ are not computationally identical.
  - We have renamed a free variable!
- $\lambda x.(zx)(zx)$ and $\lambda y.(zy)(zx)$ are not computationally identical.
  - We have to rename consistently!

Variable renaming and α-equivalence

- We will say that terms that differ only in the names of their bound variables are α-**equivalent** (computationally equivalent).
  - $M =_\alpha N$ iff M and N are α-equivalent.
  - Terms that are α-equivalent will be considered the same term.

α-equivalence exercises

- Are the following terms α-equivalent?
  - $\lambda x.xyx$ and $\lambda z.zyz$ **Yes**
  - $\lambda y.xy$ and $\lambda z.yz$ **No**
  - $(\lambda x.zx)(\lambda y.zy)$ and $(\lambda y.zy)(\lambda x.xz)$ **No**
  - $(\lambda x.x)z$ and $(\lambda z.z)z$ **Yes**
  - $(\lambda x.x)z$ and $(\lambda z.z)x$ **No**

Checked

Variable capture and α-equivalence

- Variable renaming should preserve the meaning of the term.
- If we rename a variable in such a way that a variable that was free before but is bound after, then we say that variable has been **captured**.
  - *e.g.* renaming x as y in the term $\lambda y.xy$ will capture the variable.
- If we capture a variable when renaming then we will not preserve the meaning of the term.
  - Therefore, we aim to avoid capturing variables when renaming.
- Capturing variables can be avoided with α-equivalences.
  - *e.g.* $\lambda y.xy =_\alpha \lambda z.xz$, we can now safely rename x as y without captured variables.

Computation

- Computation in the Lambda Calculus is composed of a series of substitution rewritings, known as β-**reductions**.
- A **redex** is a term of the form $(\lambda x.M)N$.
- Redexes can be β-reduced.

The β-reduction rule

- **The β-reduction rule:**

$$(\lambda x.M)N \to_\beta M\{x \mapsto N\}$$

  where $M\{x \mapsto N\}$ is the term obtained when we substitute x by N *taking into account bound variables*.

- We can apply the β-reduction rule to any redex in a term, it does not have to be at the start. The redex can be a subterm.
- If $M \to_\beta M_1 \to_\beta M_2 \to_\beta ... \to_\beta M_n$ then we write $M \to_\beta^* M_n$

β-reduction exercises

- Apply a single β-reduction to the following terms.
  - $(\lambda x.x)y$ **y**
  - $(\lambda x.xx)(\lambda xyz.(xy)(xz)(yx)(yz))$ **(λxyz.(xy)(xz)(yx)(yz))(λxyz.(xy)(xz)(yx)(yz))**
  - $(\lambda z.yz)(\lambda x.xz)$ **y(λx.xz)**
  - $(\lambda x.xxx)((\lambda y.y)z)$ **((λy.y)z)((λy.y)z)((λy.y)z)**
    weak head?

Checked

Normal forms

- When should we stop applying β-reductions?
- **Normal form:** stop when there are no more redexes left to reduce.
- **Weak-head normal form:** stop when all redexes are under an abstraction.

- β-reductions are **confluent**.
  - If $M \to_\beta M_1$ and $M \to_\beta M_2$ then there exists a term M_3 such that $M_1 \to_{\beta^*} M_3$ and $M_2 \to_{\beta^*} M_3$.
  - It doesn't matter which order you apply β-reductions, you will always reach the same normal form or weak-head normal form.

Normal form exercises

- β-reduce the following terms to their normal forms and weak-head normal forms.
  - $(\lambda x.xxx)((\lambda y.y)z)$
  - $\lambda abc.(\lambda x.a(\lambda y.xy))bc$ **already in weak head normal form** **λabc.a(bc) normal form**
  - $(\lambda x.xx)(\lambda x.xx)$ **(λx.xx)(λx.xx) infinite loop**

Checked

Arithmetic in Lambda Calculus: numbers

- To represent numbers and arithmetic we don't need to introduce digits or additional operators.
- The natural numbers can be represented with the **Church Numerals**

$$
\begin{aligned}
\overline{0} \quad &= \lambda x.\lambda y.y \\
\overline{1} \quad &= \lambda xy.xy \\
\overline{2} \quad &= \lambda xy.x(xy) \\
\overline{3} \quad &= \lambda xy.x(x(xy))y \\
\ldots
\end{aligned}
$$

this y shouldn't be here

We use $\overline{n}$ to denote the Church Numeral representing the number n

Arithmetic in Lambda Calculus: successor function

- We can define the successor function — the function that takes a number $\overline{n}$ and returns $\overline{n+1}$.

$$S = \lambda xyz.y(xyz)$$

- *e.g.* $S(\overline{0})$
    - $(\lambda xyz.y(xyz))(\lambda xy.y) \rightarrow_\beta$
      $\lambda yz.y((\lambda xy.y)yz) \rightarrow_\beta$
      $\lambda yz.y(z) =_\alpha \lambda xy.x(y) = \overline{1}$

Arithmetic in Lambda Calculus: addition

- We can define addition.

$$ADD = \lambda xyab.(xa)(yab)$$

- **Exercise:** check this works for $2 + 3$ and $1 + 0$.

Booleans in Lambda Calculus

- Boolean values and operators can also be encoded in pure Lambda Calculus.

$$\mathsf{FALSE} = \lambda xy.y$$
$$\mathsf{TRUE} = \lambda xy.x$$

- We can define the NOT operation as follows:

$$\mathsf{NOT} = \lambda x.x \ \mathsf{FALSE} \ \mathsf{TRUE}$$

- **Exercise:** check NOT works for the terms NOT TRUE and NOT FALSE.
- **Challenge:** define AND in the Lambda Calculus

Functional programming

- The Lambda Calculus is the theoretical basis for **functional programming**.
  - *e.g.* Haskel, Lisp, WolframAlpha.
  - Java Lambdas!
- The Lambda Calculus and Turing Machines are computationally equivalent
  - Proof: they can simulate each other.
- Next, we look at another model of computation, this time based on **interaction**.