5. Introduction to the Lambda Calculus

Oscar Nierstrasz
Roadmap

- What is Computability? — Church’s Thesis
- Lambda Calculus — operational semantics
- The Church-Rosser Property
- Modelling basic programming constructs


Roadmap

> What is Computability? — Church’s Thesis
> Lambda Calculus — operational semantics
> The Church-Rosser Property
> Modelling basic programming constructs
What is Computable?

Computation is usually modelled as a *mapping from inputs to outputs*, carried out by a formal “machine,” or program, which processes its input in a *sequence of steps*.

An “effectively computable” function is one that can be computed in a *finite amount of time using finite resources*.
Church’s Thesis

Effectively computable functions [from positive integers to positive integers] are just those definable in the lambda calculus.

Or, equivalently:

It is not possible to build a machine that is more powerful than a Turing machine.

Church’s thesis cannot be proven because “effectively computable” is an intuitive notion, not a mathematical one. It can only be refuted by giving a counter-example — a machine that can solve a problem not computable by a Turing machine.

So far, all models of effectively computable functions have shown to be equivalent to Turing machines (or the lambda calculus).
Uncomputability

A problem that cannot be solved by any Turing machine in finite time (or any equivalent formalism) is called uncomputable.

Assuming Church’s thesis is true, an uncomputable problem cannot be solved by any real computer.

The Halting Problem:
Given an arbitrary Turing machine and its input tape, will the machine eventually halt?

The Halting Problem is provably uncomputable — which means that it cannot be solved in practice.
What is a Function? (I)

**Extensional view:**

A (total) function $f: A \rightarrow B$ is a subset of $A \times B$ (i.e., a relation) such that:

1. for each $a \in A$, there exists some $(a, b) \in f$ (i.e., $f(a)$ is defined), and

2. if $(a, b_1) \in f$ and $(a, b_2) \in f$, then $b_1 = b_2$ (i.e., $f(a)$ is unique)
What is a Function? (II)

**Intensional view:**

A function \( f: A \rightarrow B \) is an *abstraction* \( \lambda x.e \), where \( x \) is a *variable name*, and \( e \) is an *expression*, such that when a value \( a \in A \) is *substituted* for \( x \) in \( e \), then this expression (i.e., \( f(a) \)) evaluates to some (unique) value \( b \in B \).
Roadmap

- What is Computability? — Church’s Thesis
- Lambda Calculus — operational semantics
- The Church-Rosser Property
- Modelling basic programming constructs
What is the Lambda Calculus?

The Lambda Calculus was invented by Alonzo Church [1932] as a mathematical formalism for expressing computation by functions.

Syntax:

\[ e ::= x \quad a \text{ variable} \]
\[ l \lambda x . e \quad an \ abstraction \ (function) \]
\[ l e_1 e_2 \quad a \ (function) \ application \]

Examples:

\[ \lambda x . x \] — is a function taking an argument \( x \), and returning \( x \)
\[ f x \] — is a function \( f \) applied to an argument \( x \)

**NB:** same as \( f(x) \)!
**Parsing Lambda Expressions**

**Lambda extends as far as possible to the right**

\[ \lambda f.x \ y \quad \equiv \quad \lambda f.(x \ y) \]

**Application is left-associative**

\[ x \ y \ z \quad \equiv \quad (x \ y) \ z \]

**Multiple lambdas may be suppressed**

\[ \lambda f \ g.x \quad \equiv \quad \lambda f.\lambda g.x \]
What is the Lambda Calculus? ...

*(Operational) Semantics:*

\[ \alpha \text{ conversion (renaming):} \quad \lambda x . e \leftrightarrow \lambda y . [y/x] e \quad \text{where } y \text{ is not free in } e \]

\[ \beta \text{ reduction (application):} \quad (\lambda x . e_1) e_2 \rightarrow [e_2/x] e_1 \quad \text{avoiding name capture} \]

\[ \eta \text{ reduction:} \quad \lambda x . e x \rightarrow e \quad \text{if } x \text{ is not free in } e \]

The lambda calculus can be viewed as the simplest possible pure functional programming language.
Beta Reduction

Beta reduction is the *computational engine* of the lambda calculus:

Define: \( I \equiv \lambda x . x \)

Now consider:

\[
I I = (\lambda x . x) (\lambda x . x) \rightarrow [\lambda x . x / x] x \quad \beta \text{ reduction}
\]

\[
= \lambda x . x \quad \text{substitution}
\]

\[
= I
\]
We can implement most lambda expressions directly in Haskell:

\[
i = \lambda x \to x
\]

? i 5
5

(2 reductions, 6 cells)

? i i 5
5

(3 reductions, 7 cells)
Lambdas are anonymous functions

A lambda abstraction is just an *anonymous function*.

Consider the Haskell function:

\[
\text{compose } f \ g \ x = f(g(x))
\]

The *value* of \texttt{compose} is the anonymous lambda abstraction:

\[
\lambda f \ g \ x . f(g(x))
\]

*NB: This is the same as:*

\[
\lambda f \ . \lambda g . \lambda x . f(g(x))
\]
A Few Examples

1. \((\lambda x.x) \ y\)
2. \((\lambda x.f \ x)\)
3. \(x \ y\)
4. \((\lambda x.x) \ (\lambda x.x)\)
5. \((\lambda x.x \ y) \ z\)
6. \((\lambda x.y.x) \ t \ f\)
7. \((\lambda x.y.z.z \ x \ y) \ a \ b \ (\lambda x.y.x)\)
8. \((\lambda f.g.f \ g) \ (\lambda x.x) \ (\lambda x.x) \ z\)
9. \((\lambda x.y.x \ y) \ y\)
10. \((\lambda x.y.x \ y) \ (\lambda x.x) \ (\lambda x.x)\)
11. \((\lambda x.y.x \ y) \ ((\lambda x.x) \ (\lambda x.x))\)
Free and Bound Variables

The variable $x$ is **bound** by $\lambda$ in the expression: $\lambda x.e$

A variable that is not bound, is **free**:

$$
\text{fv}(x) = \{ x \} \\
\text{fv}(e_1 e_2) = \text{fv}(e_1) \cup \text{fv}(e_2) \\
\text{fv}(\lambda x . e) = \text{fv}(e) - \{ x \}
$$

An expression with no free variables is **closed**.  
(AKA a **combinator**.) Otherwise it is **open**.

For example, $y$ is **bound** and $x$ is **free** in the (open) expression: $\lambda y . x y$
“Hello World” in the Lambda Calculus

hello world

Is this expression open? Closed?
Roadmap

> What is Computability? — Church’s Thesis
> Lambda Calculus — operational semantics
> The Church-Rosser Property
> Modelling basic programming constructs
Why macro expansion is wrong

**Syntactic substitution will not work:**

$$(\lambda x y . x y) y \rightarrow [y/x](\lambda y . x y) \overset{\beta \text{ reduction}}{\neq} (\lambda y . y y)$$

Since $y$ is *already bound* in $(\lambda y . x y)$, we cannot directly substitute $y$ for $x$. 
We must define substitution carefully to avoid *name capture*:

\[
\begin{align*}
[e/x] x &= e \\
[e/x] y &= y & \text{if } x \neq y \\
[e/x] (e_1 e_2) &= ([e/x] e_1) ([e/x] e_2) \\
[e/x] (\lambda x . e_1) &= (\lambda x . e_1) \\
[e/x] (\lambda y . e_1) &= (\lambda y . [e/x] e_1) & \text{if } x \neq y \text{ and } y \not\in \text{fv}(e) \\
[e/x] (\lambda y . e_1) &= (\lambda z . [e/x] [z/y] e_1) & \text{if } x \neq y \text{ and } \\
&\quad z \not\in \text{fv}(e) \cup \text{fv}(e_1)
\end{align*}
\]

**Consider:**

\[
(\lambda x . ((\lambda y . x) (\lambda x . x))) x \ y \rightarrow [y / x] ((\lambda y . x) (\lambda x . x)) x \\
= ((\lambda z . y) (\lambda x . x)) y
\]
Alpha conversions allow us to rename bound variables. A bound name $x$ in the lambda abstraction $\lambda x.e$ may be substituted by any other name $y$, as long as there are no free occurrences of $y$ in $e$:

Consider:

$$(\lambda x y . x y) y \rightarrow (\lambda x z . x z) y \quad \alpha \text{ conversion}$$

$$\rightarrow [y / x] (\lambda z . x z) \quad \beta \text{ reduction}$$

$$\rightarrow (\lambda z . y z)$$

$$= y \quad \eta \text{ reduction}$$
Eta Reduction

Eta reductions allow one to remove “redundant lambdas”.

Suppose that \( f \) is a \textit{closed expression} (i.e., there are no free variables in \( f \)).

Then:

\[
(\lambda x . f x) y \rightarrow f y \quad \beta \text{ reduction}
\]

So, \((\lambda x . f x)\) behaves the same as \( f \)!

Eta reduction says, \textit{whenever \( x \) does not occur free in \( f \), we can rewrite \((\lambda x . f x)\) as \( f \).}
\[ (\lambda x y . x y) (\lambda x . x y) (\lambda a b . a b) \quad \text{NB: left assoc.} \]
\[ \rightarrow (\lambda x z . x z) (\lambda x . x y) (\lambda a b . a b) \quad \alpha \text{ conversion} \]
\[ \rightarrow (\lambda z . (\lambda x . x y) z) (\lambda a b . a b) \quad \beta \text{ reduction} \]
\[ \rightarrow (\lambda x . x y) (\lambda a b . a b) \quad \beta \text{ reduction} \]
\[ \rightarrow (\lambda a b . a b) y \quad \beta \text{ reduction} \]
\[ \rightarrow (\lambda b . y b) \quad \beta \text{ reduction} \]
\[ \rightarrow y \quad \eta \text{ reduction} \]
Normal Forms

A lambda expression is in **normal form** if it can no longer be reduced by *beta or eta reduction rules*.

Not all lambda expressions have normal forms!

$$\Omega = (\lambda x . x x) (\lambda x . x x)$$

$$\rightarrow [ (\lambda x . x x) / x ] ( x x )$$

$$= (\lambda x . x x) (\lambda x . x x) \quad \beta \text{ reduction}$$

$$\rightarrow (\lambda x . x x) (\lambda x . x x) \quad \beta \text{ reduction}$$

$$\rightarrow (\lambda x . x x) (\lambda x . x x) \quad \beta \text{ reduction}$$

$$\rightarrow \ldots$$

Reduction of a lambda expression to a normal form is analogous to a *Turing machine halting* or a *program terminating*. 
Most programming languages are strict, that is, all expressions passed to a function call are evaluated before control is passed to the function. Most modern functional languages, on the other hand, use lazy evaluation, that is, expressions are only evaluated when they are needed.

Consider: \( \text{sqr} \ n = n \times n \)

Applicative-order reduction:

\[
\text{sqr} \ (2+5) \ \leftrightarrow \ \text{sqr} \ 7 \ \leftrightarrow \ 7 \times 7 \ \leftrightarrow \ 49
\]

Normal-order reduction:

\[
\text{sqr} \ (2+5) \ \leftrightarrow \ (2+5) \times (2+5) \ \leftrightarrow \ 7 \times (2+5) \ \leftrightarrow \ 7 \times 7 \ \leftrightarrow \ 49
\]
The Church-Rosser Property

“If an expression can be evaluated at all, it can be evaluated by consistently using normal-order evaluation. If an expression can be evaluated in several different orders (mixing normal-order and applicative order reduction), then all of these evaluation orders yield the same result.”

So, evaluation order “does not matter” in the lambda calculus.
Roadmap

- What is Computability? — Church’s Thesis
- Lambda Calculus — operational semantics
- The Church-Rosser Property
- Modelling basic programming constructs
Non-termination

However, applicative order reduction may not terminate, even if a normal form exists!

\[(\lambda x . y) \ ( (\lambda x . x x) \ (\lambda x . x x) ) \]

**Applicative order reduction**
\[
\rightarrow (\lambda x . y) \ ( (\lambda x . x x) \ (\lambda x . x x) )
\]
\[
\rightarrow (\lambda x . y) \ ( (\lambda x . x x) \ (\lambda x . x x) )
\]

... 

**Normal order reduction**
\[
\rightarrow y
\]

Compare to the Haskell expression:

\[
(\x \rightarrow \y \rightarrow x) \ 1 \ (5/0) \leftrightarrow 1
\]
Currying

Since a lambda abstraction only binds a single variable, functions with multiple parameters must be modelled as Curried higher-order functions.

As we have seen, to improve readability, multiple lambdas are suppressed, so:

\[ \lambda x \, y . \, x = \lambda x . \lambda y . \, x \]
\[ \lambda b \, x \, y . \, b \, x \, y = \lambda b . \lambda x . \lambda y . \,( b \, x ) \, y \]
Many programming concepts can be directly expressed in the lambda calculus. Let us define:

\begin{align*}
  \text{True} & \equiv \lambda x y . x \\
  \text{False} & \equiv \lambda x y . y \\
  \text{not} & \equiv \lambda b . b \text{False True} \\
  \text{if } b \text{ then } x \text{ else } y & \equiv \lambda b x y . b x y
\end{align*}

then:

\begin{align*}
  \text{not True} & = (\lambda b . b \text{False True}) (\lambda x y . x) \\
  & \rightarrow (\lambda x y . x) \text{False True} \\
  & \rightarrow \text{False} \\
  \text{if True then } x \text{ else } y & = (\lambda b x y . b x y) (\lambda x y . x) x y \\
  & \rightarrow (\lambda x y . x) x y \\
  & \rightarrow x
\end{align*}
Representing Tuples

Although tuples are not supported by the lambda calculus, they can easily be modelled as higher-order functions that “wrap” pairs of values. n-tuples can be modelled by composing pairs ...

Define:

\[
\begin{align*}
\text{pair} & \equiv (\lambda \, x \, y \, z . \, z \, x \, y) \\
\text{first} & \equiv (\lambda \, p . \, p \, \text{True}) \\
\text{second} & \equiv (\lambda \, p . \, p \, \text{False})
\end{align*}
\]

then:

\[
\begin{align*}
(1, \, 2) & = \text{pair} \, 1 \, 2 \\
\rightarrow & \quad (\lambda \, z . \, z \, 1 \, 2) \\
\text{first} \, (\text{pair} \, 1 \, 2) & \rightarrow (\text{pair} \, 1 \, 2) \, \text{True} \\
\rightarrow & \quad \text{True} \, 1 \, 2 \\
\rightarrow & \quad 1
\end{align*}
\]
Tuples as functions

In Haskell:

\[
\begin{align*}
t & = \lambda x \rightarrow \lambda y \rightarrow x \\
f & = \lambda x \rightarrow \lambda y \rightarrow y \\
pair & = \lambda x \rightarrow \lambda y \rightarrow \lambda z \rightarrow z \, x \, y \\
\text{first} & = \lambda p \rightarrow p \, t \\
\text{second} & = \lambda p \rightarrow p \, f
\end{align*}
\]

? first (pair 1 2)  
1
? first (second (pair 1 (pair 2 3)))  
2
What you should know!

✎ Is it possible to write a Pascal compiler that will generate code just for programs that terminate?
✎ What are the alpha, beta and eta conversion rules?
✎ What is name capture? How does the lambda calculus avoid it?
✎ What is a normal form? How does one reach it?
✎ What are normal and applicative order evaluation?
✎ Why is normal order evaluation called lazy?
✎ How can Booleans and tuples be represented in the lambda calculus?
Can you answer these questions?

✎ How can name capture occur in a programming language?
✎ What happens if you try to program Ω in Haskell? Why?
✎ What do you get when you try to evaluate (pred 0)? What does this mean?
✎ How would you model numbers in the lambda calculus? Fractions?
Attribution-ShareAlike 3.0

You are free:
- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:

**BY:** Attribution. You must attribute the work in the manner specified by the author or licensor.

**Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

http://creativecommons.org/licenses/by-sa/3.0/