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**).
The extensional view is the database view: a function is a particular set of mappings from arguments to values.
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 \( b \in B \).
The intensional view is the programmatic view: a function is a \textit{specification} of how to transform the input argument to an output value.

NB: uniqueness does not come for free. The latter view is closer to that of programming languages, since infinite relations can only be represented intensionally.
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:**

\[
\begin{align*}
e & ::= x \quad \text{a variable} \\
    & | \lambda x . e \quad \text{an abstraction (function)} \\
    & | e_1 e_2 \quad \text{a (function) application}
\end{align*}
\]

**Examples:**

\(
\lambda x . x \quad \text{is a function taking an argument } x, \text{ and returning } x
\)

\(f x \quad \text{is a function } f \text{ applied to an argument } x\)

**NB:** *same as } f(x) !*
We have seen lambda abstractions before in Haskell with a very similar syntax:

\( \lambda \ x \rightarrow x + 1 \)

is the anonymous Haskell function that adds 1 to its argument \( x \). Function application in Haskell also has the same syntax as in the lambda calculus:

```
Prelude> (\x -> x+1) 2
3
```
Lambda extends as far as possible to the right
\[ \lambda f. x \ y \equiv \lambda f. (x \ y) \]

Application is left-associative
\[ x \ y \ z \equiv (x \ y) \ z \]

Multiple lambdas may be suppressed
\[ \lambda f \ g. x \equiv \lambda f. \ \lambda g. x \]
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.
The **α conversion** rule simply states that “*variable names don't matter*”. If you define a function with an argument \( x \), you can change the name of \( x \) to \( y \), as long as you do it consistently (change every \( x \) to \( y \)) and avoid name clashes (there must not be another \( y \) in the same scope).

The **β reduction** rule shows *how to evaluate function application*: just (syntactically) replace the formal parameter of the function body by the argument everywhere, taking care to avoid name clashes.

Finally, the **η reduction** rule can be seen as a *renaming optimization*: if the body of a function just applies another function \( f \) to its argument, then we can replace that whole function by \( f \).

Note that the **α** rule only rewrites an expression but does not simplify it. That is why it is called a “conversion” and not a “reduction”.
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 \text{\(\beta\) reduction} \]
\[ = \lambda x . x \quad \text{substitution} \]
\[ = I \]
In the expression:

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

we replace the x in the body of the first lambda by its argument. The body is simply x, so we end up with \((\lambda x . x)\)

Let's number each x to make clear what is happening:

\[(\lambda x_1 . x_2) (\lambda x_3 . x_4)\]

\(x_1\) and \(x_3\) are formal parameters, and \(x_2\) and \(x_4\) are the bodies of the two lambda expressions. We are applying the first expression \((\lambda x_1 . x_2)\) as a function to its argument \((\lambda x_3 . x_4)\)

To do this, we replace the body of \((\lambda x_1 . x_2)\), i.e., \(x_2\), by the argument \((\lambda x_3 . x_4)\). This is written as follows:

\[\left[ (\lambda x_3 . x_4) / x_2 \right] x_2\]

This leaves as the end result: \((\lambda x_3 . x_4)\) (i.e., \((\lambda x . x)\)).
Lambda expressions in Haskell

We can implement many lambda expressions directly in Haskell:

```haskell
Prelude> let i = \x -> x
Prelude> i 5
5
Prelude> i i 5
5
```

How is `i i 5` parsed?
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 `compose` is the anonymous lambda abstraction:

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

*NB: This is the same as:*

\[ \lambda f . \lambda g . \lambda x . f(g(x)) \]
Prelude> let compose = \f g x -> f(g x)
Prelude> compose (\x->x+1) (\x->x*2) 5
11
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**:

\[
\begin{align*}
fv(x) &= \{ x \} \\
fv(e_1 e_2) &= fv(e_1) \cup fv(e_2) \\
fv(\lambda x . e) &= fv(e) - \{ x \}
\end{align*}
\]

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:
\( \lambda y . x y \)
You can also think of *bound* variables as being *defined*. The expression

\[ \lambda x. e \]

defines the variable \( x \) within the body \( e \), just like:

```java
int plus(int x, int y) { ... }
```

defines the variables \( x \) and \( y \) within the body of the Java method \( plus \).

A variable that is not defined in some outer scope by some lambda is “*free*”, or simply *undefined*. Closed expressions have no “undefined” variables. In statically typed programming languages, all procedures and programs are normally closed.
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. \((\lambda x.y.x\ y)\ ((\lambda x.x)\ (\lambda x.x))\)

Which variables are free? Which are bound?
“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:**

\[(\lambda x \cdot x \, y \, y) \, y \rightarrow [y/x] (\lambda y \cdot x \, y) \quad \beta \text{ reduction} \]

\[\neq (\lambda y \cdot y \, y \, y) \quad \text{incorrect substitution!} \]

Since y is *already bound* in \((\lambda y \cdot x \, y)\), we cannot directly substitute y for x.
Substitution

We must define substitution carefully to avoid *name capture*:

\[
\begin{align*}
[e/x] x &= e \\
[e/x] y &= y & \text{if } x &\neq y \\
[e/x] (e_1 e_2) &= ([e/x] e_1) ([e/x] e_2) \\
[e/x] (\lambda x . e_1) &= (\lambda x . e_1) \\
[e/x] (\lambda y . e_1) &= (\lambda y . [e/x] e_1) & \text{if } x &\neq y \text{ and } y \notin \text{fv}(e) \\
[e/x] (\lambda y . e_1) &= (\lambda z . [e/x] [z/y] e_1) & \text{if } x &\neq y \text{ and } z \notin \text{fv}(e) \cup \text{fv}(e_1)
\end{align*}
\]

Consider:

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

\[
= ((\lambda z . y) (\lambda x . x)) y
\]
Of these six cases, only the last one is tricky.

If the expression e (i.e., the argument to our function \((\lambda y . e_1)\)) contains a variable name y that conflicts with the formal parameter y of our function, then we must first rename y to a fresh name z in that function. After renaming y to z, there is no longer any conflict with the name y in our argument e, and we can proceed safely with the substitution.
Alpha Conversion

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:

$$(\lambda x \, y \, . \, x \, y) \, y \quad \rightarrow \quad (\lambda x \, z \, . \, x \, z) \, y \quad \alpha \ \text{conversion}$$
$$\rightarrow \ [ y / x] \ (\lambda z \, . \, x \, z) \quad \beta \ \text{reduction}$$
$$\rightarrow \ (\lambda z \, . \, y \, z)$$
$$= \ y \quad \eta \ \text{reduction}$$
Eta Reduction

Eta reductions allow one to remove “redundant lambdas”.
Suppose that \( f \) is a *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, *whenever \( x \) does not occur free in \( f \), we can rewrite \((\lambda x . f x)\) as \( f \).*
$\alpha \beta \eta$


\[
\begin{align*}
(\lambda \ x \ y \ . \ x \ y) & \ (\lambda \ x \ . \ x \ y) \ (\lambda \ a \ b \ . \ a \ b) & NB: \ left \ assoc. \\
\rightarrow & \ (\lambda \ x \ z \ . \ x \ z) \ (\lambda \ x \ . \ x \ y) \ (\lambda \ a \ b \ . \ a \ b) & \alpha \ conversion \\
\rightarrow & \ (\lambda \ z \ . \ (\lambda \ x \ . \ x \ y) \ z) \ (\lambda \ a \ b \ . \ a \ b) & \beta \ reduction \\
\rightarrow & \ (\lambda \ x \ . \ x \ y) \ (\lambda \ a \ b \ . \ a \ b) & \beta \ reduction \\
\rightarrow & \ (\lambda \ a \ b \ . \ a \ b) \ y & \beta \ reduction \\
\rightarrow & \ (\lambda \ b \ . \ y \ b) & \beta \ reduction \\
\rightarrow & \ y & \eta \ reduction 
\end{align*}
\]
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!

\[ \Omega = (\lambda x . x \ x) (\lambda x . x \ x) \]

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

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

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

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

\[\rightarrow \cdots\]

Reduction of a lambda expression to a normal form is analogous to a *Turing machine halting* or a *program terminating.*
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. \((\lambda x. y. x\ y)\) \(((\lambda x. x)(\lambda x. x))\)

Are these in normal form? Can they be reduced? If so, how?
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:

```
sqr n = n * n
```

Applicative-order reduction:

```
sqr (2+5) ⇝ sqr 7 ⇝ 7*7 ⇝ 49
```

Normal-order reduction:

```
sqr (2+5) ⇝ (2+5) * (2+5) ⇝ 7 * (2+5) ⇝ 7 * 7 ⇝ 49
```
“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!

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

Applicative order reduction  Normal order reduction

\[\rightarrow (\lambda x \cdot y) \ ( (\lambda x \cdot x \ x) \ (\lambda x \cdot x \ x) )\]

\[\rightarrow (\lambda x \cdot y) \ ( (\lambda x \cdot x \ x) \ (\lambda x \cdot x \ x) )\]

\[\ldots\]

Compare to the Haskell expression:

\[(\x \rightarrow \ y \rightarrow \ x) \ 1 \ (5/0) \ \Rightarrow \ 1\]
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:

\[
\lambda x y . x = \lambda x . \lambda y . x \\
\lambda b x y . b x y = \lambda b . \lambda x . \lambda y . (b x) y
\]
Don’t forget that functions written this way are still Curried, so arguments can be bound one at a time!

In Haskell:

```
Prelude> let f = (\ x y -> x) 1
Prelude> f 2
1
```
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:**

\[
\begin{align*}
\text{not True} & = (\lambda b . \ b \ \text{False} \ True ) (\lambda x \ y . \ x ) \\
& \rightarrow (\lambda x \ y . \ x ) \ \text{False} \ True \\
& \rightarrow \text{False} \\
\text{if True then } x \ \text{else } y & = (\lambda b x y . \ b x y ) (\lambda x y . \ x) \ x \ y \\
& \rightarrow (\lambda x y . \ x) \ x \ y \\
& \rightarrow x
\end{align*}
\]
This is the “standard encoding” of Booleans as lambdas (other encodings are possible).

A Boolean makes a choice between two values, a “true” one and a “false” one. True returns the first argument and False returns the second.

Negation just reverses the logic, by passing False and True as arguments to the boolean: not True will return False and not False will return True.
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:**

- pair  ≡  (\( \lambda \ x \ y \ z \ . \ z \ x \ y \))
- first  ≡  (\( \lambda \ p \ . \ p \ True \))
- second  ≡  (\( \lambda \ p \ . \ p \ False \))

**then:**

- (1, 2)  ≡  pair 1 2
  →  (\( \lambda \ z \ . \ z \ 1 \ 2 \))
  →  (pair 1 2) True
  →  True 1 2
  →  1
The function `pair` takes three arguments. The first two arguments are the x and y values of the pair. Since `pair` is a Curried function, passing in x and y returns a function (i.e., a pair) that will take a third argument, z. The body of the pair will pass x and y to z, which can then bind x and y and do what it likes with them.

As examples, consider the functions first and second. Each takes a pair p as argument and passes it a boolean as the final argument z. These booleans respectively return x or y, i.e., the first or second value in the pair.

*How would you define a lambda expression `sum` that takes a pair p as argument and returns the sum of the x and y values it contains?*
Tuples as functions

In Haskell:

```haskell
let t = \x -> \y -> x
let f = \x -> \y -> y
let pair = \x -> \y -> \z -> z x y
let first = \p -> p t
let second = \p -> p f

Prelude> first (pair 1 2)
1
Prelude> 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 $\Omega$ 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 4.0 International (CC BY-SA 4.0)

You are free to:

**Share** — copy and redistribute the material in any medium or format

**Adapt** — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

**Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

**ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

[http://creativecommons.org/licenses/by-sa/4.0/](http://creativecommons.org/licenses/by-sa/4.0/)