UCT Algorithm Circle: Lambda Calculus

Graham Manuell

21 April 2011
Introduction to Lambda Calculus

1. Introduction to Lambda Calculus
2. Untyped Lambda Calculus in Action
3. Typed Lambda Calculi
History

- In the early 1920s, Moses Schönfinkel develops *Combinatory logic* and, with it, the concept of currying.

- In the 1930s, Alonzo Church develops *Lambda Calculus* as a foundational system for mathematics.

- Both Combinatory logic and Lambda Calculus are proved inconsistent by Stephen Kleene and J.B. Rosser in 1935.

- In 1936, Church publishes his *Untyped Lambda Calculus* as a model of computation, ahead of Turing’s paper.

- In 1939, Rosser proves both models equivalent.

- Church develops the restricted, but logically consistent, system of *Simply Typed Lambda Calculus* in 1940.
History

- In the early 1920s, Moses Schönfinkel develops *Combinatory logic* and, with it, the concept of currying.
- In the 1930s, Alonzo Church develops *Lambda Calculus* as a foundational system for mathematics.
- Both Combinatory logic and Lambda Calculus are proved inconsistent by Stephen Kleene and J.B. Rosser in 1935.
- In 1936, Church publishes his *Untyped Lambda Calculus* as a model of computation, ahead of Turing’s paper.
- In 1939, Rosser proves both models equivalent.
- Church develops the restricted, but logically consistent, system of *Simply Typed Lambda Calculus* in 1940.
In the early 1920s, Moses Schönfinkel develops *Combinatory logic* and, with it, the concept of currying.

In the 1930s, Alonzo Church develops *Lambda Calculus* as a foundational system for mathematics.

Both Combinatory logic and Lambda Calculus are proved inconsistent by Stephen Kleene and J.B. Rosser in 1935.

In 1936, Church publishes his *Untyped Lambda Calculus* as a model of computation, ahead of Turing’s paper.

In 1939, Rosser proves both models equivalent.

Church develops the restricted, but logically consistent, system of *Simply Typed Lambda Calculus* in 1940.
History

- In the early 1920s, Moses Schönfinkel develops *Combinatory logic* and, with it, the concept of currying.
- In the 1930s, Alonzo Church develops *Lambda Calculus* as a foundational system for mathematics.
- Both Combinatory logic and Lambda Calculus are proved inconsistent by Stephen Kleene and J.B. Rosser in 1935.
- In 1936, Church publishes his *Untyped Lambda Calculus* as a model of computation, ahead of Turing’s paper.
- In 1939, Rosser proves both models equivalent.
- Church develops the restricted, but logically consistent, system of *Simply Typed Lambda Calculus* in 1940.
In the early 1920s, Moses Schönfinkel develops *Combinatory logic* and, with it, the concept of currying.

In the 1930s, Alonzo Church develops *Lambda Calculus* as a foundational system for mathematics.

Both Combinatory logic and Lambda Calculus are proved inconsistent by Stephen Kleene and J.B. Rosser in 1935.

In 1936, Church publishes his *Untyped Lambda Calculus* as a model of computation, ahead of Turing’s paper.

In 1939, Rosser proves both models equivalent.

Church develops the restricted, but logically consistent, system of *Simply Typed Lambda Calculus* in 1940.
History

- In the early 1920s, Moses Schönfinkel develops *Combinatory logic* and, with it, the concept of currying.
- In the 1930s, Alonzo Church develops *Lambda Calculus* as a foundational system for mathematics.
- Both Combinatory logic and Lambda Calculus are proved inconsistent by Stephen Kleene and J.B. Rosser in 1935.
- In 1936, Church publishes his *Untyped Lambda Calculus* as a model of computation, ahead of Turing’s paper.
- In 1939, Rosser proves both models equivalent.
- Church develops the restricted, but logically consistent, system of *Simply Typed Lambda Calculus* in 1940.
Lambda Calculus is based on *anonymous* functions.

- $x \mapsto x^2$ instead of $f(x) = x^2$
- Functions can be *applied* to their arguments.
  - $(x \mapsto 2 + x)(3)$ becomes $2 + 3$
- Functions can be of *higher order*.
  - they can take other functions as arguments.
- Functions of multiple arguments are *curried*.
  - $x \mapsto (y \mapsto x + y)$ instead of $(x, y) \mapsto x + y$
- The names of the arguments are (often) irrelevant.
  - $x \mapsto x$ is equivalent to $y \mapsto y$
Lambda Calculus is based on *anonymous* functions.
- $x \mapsto x^2$ instead of $f(x) = x^2$

Functions can be *applied* to their arguments.
- $(x \mapsto 2 + x)(3)$ becomes $2 + 3$

Functions can be of *higher order*.
- they can take other functions as arguments.

Functions of multiple arguments are *curried*.
- $x \mapsto (y \mapsto x + y)$ instead of $(x, y) \mapsto x + y$

The names of the arguments are (often) irrelevant.
- $x \mapsto x$ is equivalent to $y \mapsto y$
The Basics

- Lambda Calculus is based on *anonymous* functions.
  - $x \mapsto x^2$ instead of $f(x) = x^2$
- Functions can be *applied* to their arguments.
  - $(x \mapsto 2 + x)(3)$ becomes $2 + 3$
- Functions can be of *higher order*.
  - they can take other functions as arguments.
- Functions of multiple arguments are *curried*.
  - $x \mapsto (y \mapsto x + y)$ instead of $(x, y) \mapsto x + y$
- The names of the arguments are (often) irrelevant.
  - $x \mapsto x$ is equivalent to $y \mapsto y$
Lambda Calculus is based on *anonymous* functions.

- \( x \mapsto x^2 \) instead of \( f(x) = x^2 \)
- Functions can be *applied* to their arguments.
  - \( (x \mapsto 2 + x)(3) \) becomes \( 2 + 3 \)
- Functions can be of *higher order*.
  - they can take other functions as arguments.
- Functions of multiple arguments are *curried*.
  - \( x \mapsto (y \mapsto x + y) \) instead of \( (x, y) \mapsto x + y \)
- The names of the arguments are (often) irrelevant.
  - \( x \mapsto x \) is equivalent to \( y \mapsto y \)
The Basics

- Lambda Calculus is based on *anonymous* functions.
  - $x \mapsto x^2$ instead of $f(x) = x^2$

- Functions can be *applied* to their arguments.
  - $(x \mapsto 2 + x)(3)$ becomes $2 + 3$

- Functions can be of *higher order*.
  - they can take other functions as arguments.

- Functions of multiple arguments are *curried*.
  - $x \mapsto (y \mapsto x + y)$ instead of $(x, y) \mapsto x + y$

- The names of the arguments are (often) irrelevant.
  - $x \mapsto x$ is equivalent to $y \mapsto y$
A term in lambda calculus is defined as follows
- A variable, $x$, is a lambda term
- If $x$ is a variable and $t$ is a lambda term, then $\lambda x.t$ is a lambda term (*lambda abstraction*)
- If $t$ and $s$ are lambda terms, so is $t \ s$ (*application*)

The following are examples of lambda terms
- $\lambda x.x$ the identity function
- $(\lambda x.x)y$ the identity function applied to $y$
- $\lambda x.y$ the constant function
A term in lambda calculus is defined as follows
- A variable, $x$, is a lambda term.
- If $x$ is a variable and $t$ is a lambda term, then $\lambda x.t$ is a lambda term (lambda abstraction).
- If $t$ and $s$ are lambda terms, so is $t \ s$ (application).

The following are examples of lambda terms:
- $\lambda x. x$ the identity function.
- $(\lambda x. x) y$ the identity function applied to $y$.
- $\lambda x. y$ the constant function.
A Note on Notation

- Abstractions extend as far to the right as possible.
  \[ \lambda x. x \ z \ \lambda y. y \ x \text{ means } \lambda x. (x \ z \ \lambda y. (y \ x)) \]
- Multiple adjacent abstractions may be merged.
  \[ \lambda xyz. x \text{ means } \lambda x. \lambda y. \lambda z. x \]
- Application associates to the left.
  \[ w \ x \ y \ z \text{ means } ((w \ x) \ y) \ z \]
- Brackets can be used group terms differently.
  \[ \lambda x. x \ (z \ (\lambda y. z) \ x) \]
Lambda abstraction represents defining a function.

- $\lambda x. t$ is said to bind the variable $x$ in $t$.
- A variable is free if it is not bound by an abstraction.

$x$ is bound and $y$ is free in the following:

- $\lambda x. x$
- $(\lambda x. x) y$
- $\lambda x. y \ x$
Free and bound variables

- Lambda abstraction represents defining a function.
- $\lambda x. t$ is said to bind the variable $x$ in $t$.
- A variable is free if it is not bound by an abstraction.
- $x$ is bound and $y$ is free in the following:
  - $\lambda x. x$
  - $(\lambda x. x) y$
  - $\lambda x. y \ x$
Scope

- An abstraction defines a scope for its bound variable.
- A variable is refers back to the closest preceding abstraction which mentions it.

Consider the following

- \((\lambda x. x) x\)
- \(\lambda x. x \; \lambda x. x\)
- \(\lambda x. \lambda y. \lambda x. x \; y\)
Now that we know what lambda terms are, let’s see how we manipulate them.

A lambda expression can be $\alpha$-converted to another changing the names of bound variables.

- $\lambda x. x \rightarrow_{\alpha} \lambda y. y$
- $x \not\equiv_{\alpha} y$ (x and y are free)

Note that scope must be considered when doing the conversion.

- $\lambda x. \lambda x. x \rightarrow_{\alpha} \lambda y. \lambda x. x$
- $\lambda x. \lambda x. x \not\equiv_{\alpha} \lambda y. \lambda x. y$

Additionally, the renaming cannot result in a variable being captured by another abstraction.

- $\lambda x. \lambda y. x \not\equiv_{\alpha} \lambda y. \lambda y. y$
Now that we know what lambda terms are, let’s see how we manipulate them.

A lambda expression can be $\alpha$-converted to another changing the names of *bound* variables.

- $\lambda x. x \rightarrow_{\alpha} \lambda y. y$
- $x \not\equiv_{\alpha} y$ ($x$ and $y$ are free)

Note that scope must be considered when doing the conversion.

- $\lambda x. \lambda x. x \rightarrow_{\alpha} \lambda y. \lambda x . x$
- $\lambda x. \lambda x . x \not\equiv_{\alpha} \lambda y. \lambda x . y$

Additionally, the renaming cannot result in a variable being captured by another abstraction.

- $\lambda x. \lambda y . x \not\equiv_{\alpha} \lambda y . \lambda y . y$
**β-reduction**

- **β-reduction** represents function application.
- In order to define β-reduction we must first define *capture-avoiding substitution*.
- Substitution is a way of replacing variables in a lambda expression with another term.
- We write $E[x := E']$ to mean replace $x$ with $E'$ in $E$.
- We define β-reduction by $(\lambda V.E)E' \equiv_{\beta} E[V := E']$. 

Graham Manuell
Advanced Sorting
Capture-avoiding substitution

- If $s, t, r$ are expressions and $x, y$ are variables, then
  - $x[x := r] = r$
  - $y[x := r] = y$
  - $(ts)[x := r] = (t[x := r])(s[x := r])$
  - $(\lambda x.t)[x := r] = \lambda x.t$
  - $(\lambda y.t)[x := r] = \lambda y.(t[x := r])$ if $y$ is not a free variable in $r$ (freshness condition)

- If the freshness condition does not hold, the expression may first need to be $\alpha$-converted
  - $(\lambda x.y)[y := x] \not\equiv \lambda x.(y[y := x]) \equiv \lambda x.x$
  - $(\lambda x.y)[y := x] \equiv \alpha (\lambda z.y)[y := x] \equiv \lambda z.(y[y := x]) \equiv \lambda z.x$
Capture-avoiding substitution

If $s, t, r$ are expressions and $x, y$ are variables, then

- $x[x := r] = r$
- $y[x := r] = y$
- $(ts)[x := r] = (t[x := r])(s[x := r])$
- $(\lambda x.t)[x := r] = \lambda x.t$
- $(\lambda y.t)[x := r] = \lambda y.(t[x := r])$ if $y$ is not a free variable in $r$ (freshness condition)

If the freshness condition does not hold, the expression may first need to be $\alpha$-converted

- $(\lambda x.y)[y := x] \not\equiv \lambda x.(y[y := x]) \equiv \lambda x.x$
- $(\lambda x.y)[y := x] \equiv \alpha (\lambda z.y)[y := x] \equiv \lambda z.(y[y := x]) \equiv \lambda z.x$
A lambda expression is its *normal form* if it cannot be β-reduced.

We find that β-reduction is *confluent* up to α-reduction i.e. if there are multiple orders in which an expression can be reduced, all resulting forms will be α-equivalent.

β-reduction is not normalising: there are expressions without a normal form.

The normal form of an lambda expression can be though of the result its ‘execution’.

Expressions without a normal form can be thought of as ‘infinite loops‘.
Normal Forms

- A lambda expression is its *normal form* if it cannot be \( \beta \)-reduced.
- We find that \( \beta \)-reduction is *confluent* up to \( \alpha \)-reduction i.e. if there are multiple orders in which an expression can be reduced, all resulting forms will be \( \alpha \)-equivalent.
- \( \beta \)-reduction is not normalising: there are expressions without a normal form.
- The normal form of an lambda expression can be thought of the result its ‘execution’.
- Expressions without a normal form can be thought of as ‘infinite loops‘.
While $\beta$-reduction is confluent, the order of application does affect number of iterations required to reach a normal form and even whether a normal form will be reached at all. In order to make stronger guarantees about termination we define systematic reduction strategies.
Reduction Strategies

- **Applicative order or eager evaluation**
  - The leftmost, innermost redex is reduced first.
  - Arguments are evaluated before being passed to functions.
  - Used in Lisp, ML and imperative languages.

- **Normal order**
  - The leftmost, outermost redex is reduced first.
  - Arguments are substituted before they are evaluated.

- **Call by need or lazy evaluation**
  - Normal order where duplicated work is avoided.
  - Use of thunks allows redexes to be evaluated as needed.
  - Used in Haskell and automated theorem provers.
Reduction Strategies

- **Applicative order or eager evaluation**
  - The leftmost, innermost redex is reduced first.
  - Arguments are evaluated before being passed to functions.
  - Used in Lisp, ML and imperative languages.

- **Normal order**
  - The leftmost, outermost redex is reduced first.
  - Arguments are substituted before they are evaluated.

- **Call by need or lazy evaluation**
  - Normal order where duplicated work is avoided.
  - Use of thunks allows redexes to be evaluated as needed.
  - Used in Haskell and automated theorem provers.
**Reduction Strategies**

- **Applicative order or eager evaluation**
  - The leftmost, innermost redex is reduced first.
  - Arguments are evaluated before being passed to functions.
  - Used in Lisp, ML and imperative languages.

- **Normal order**
  - The leftmost, outermost redex is reduced first.
  - Arguments are substituted before they are evaluated.

- **Call by need or lazy evaluation**
  - Normal order where duplicated work is avoided.
  - Use of thunks allows redexes to be evaluated as needed.
  - Used in Haskell and automated theorem provers.
Normal Order

- We will consider normal order evaluation.
- It is called normal order because it is a *normalising* strategy – if an expression has a normal form, it will find it.
- In contrast applicative order is *not* normalising. (However, it is possible to convert expressions into forms that applicative order can handle.)
Lambda Expressions as Programmes

- As has been hinted to for much of the lecture, lambda calculus can be thought of as a programming language.

- Lambda expressions are the code and reductions represent evaluation.

- For an expression to be meaningful as a programme it must have no free variables.

- In this language, the only data-type is functions. All others must be constructed from these.

- Purely functional languages like Haskell can simply be thought of as lambda calculus with lots of syntactic sugar. (Actually, Haskell is a typed lambda calculus – more on this later.)
Lambda Expressions as Programmes

- As has been hinted to for much of the lecture, lambda calculus can be thought of as a programming language.
- Lambda expressions are the code and reductions represent evaluation.
- For an expression to be meaningful as a programme it must have no free variables.
- In this language, the only data-type is functions. All others must be constructed from these.
- Purely functional languages like Haskell can simply be thought of as lambda calculus with lots of syntactic sugar. (Actually, Haskell is a typed lambda calculus – more on this later.)
Lambda Expressions as Programmes

- As has been hinted to for much of the lecture, lambda calculus can be thought of as a programming language.
- Lambda expressions are the code and reductions represent evaluation.
- For an expression to be meaningful as a programme it must have no free variables.
- In this language, the only data-type is functions. All others must be constructed from these.
- Purely functional languages like Haskell can simply be thought of as lambda calculus with lots of syntactic sugar. (Actually, Haskell is a typed lambda calculus – more on this later.)
Outline

1. Introduction to Lambda Calculus
2. Untyped Lambda Calculus in Action
3. Typed Lambda Calculi
Church devised a convenient way of expressing natural numbers in lambda calculus.

- 0 := λfx.x
- 1 := λfx.f x
- 2 := λfx.f (f x)
- 3 := λfx.f (f (f x))
- ...

Thus, the number $n$ is a function that takes a function $f$ and returns the $n$-th power of $f$, $f^{(n)}$.
The successor function applies \( f \) once more than \( n \) did

\[
\text{SUCC} := \lambda nfx. f (n f x)
\]

PLUS composes \( f^{(m)} \) with \( f^{(n)} \) or calls SUCC \( m \) times

\[
\text{PLUS} := \lambda mnfx. m f (n f x)
\]

\[
\text{PLUS} := \lambda mn. m \text{SUCC} n
\]

Similarly

\[
\text{MULT} := \lambda mnf. m (n f)
\]

\[
\text{MULT} := \lambda mn. m (\text{PLUS} n) \ 0
\]

\[
\text{EXP} := \lambda be. e \ b
\]

We will leave the definition of the predecessor function \( \text{PRED} \) and subtraction \( \text{SUB} \) for later, but note they are defined such that negative results become 0.
The successor function applies $f$ once more than $n$ did
- $\text{SUCC} := \lambda nfx.f \ (n \ f \ x)$

PLUS composes $f^{(m)}$ with $f^{(n)}$ or calls SUCC $m$ times
- $\text{PLUS} := \lambda mnfxf.m \ f \ (n \ f \ x)$
- $\text{PLUS} := \lambda mn.m \ \text{SUCC} \ n$

Similarly
- $\text{MULT} := \lambda mnf.m \ (n \ f)$
- $\text{MULT} := \lambda mn.m \ (\text{PLUS} \ n) \ 0$
- $\text{EXP} := \lambda be.e \ b$

We will leave the definition of the predecessor function $\text{PRED}$ and subtraction $\text{SUB}$ for later, but note they are defined such that negative results become 0.
We can also define booleans in lambda calculus.

- **TRUE** := \( \lambda xy. x \)
- **FALSE** := \( \lambda xy. y \)

The Church booleans are simply functions that take two arguments and return either the first or the second.

Interestingly, **FALSE** is equivalent to 0 up to \( \alpha \)-conversion.
Operations on Booleans

- The booleans operations are straight-forward.
  - \( \text{AND} := \lambda pq. p \ q \ p \)
  - \( \text{OR} := \lambda pq. p \ p \ q \)
  - \( \text{NOT} := \lambda pab. p \ b \ a \)
  - \( \text{IFTHENELSE} := \lambda pab. p \ a \ b \)

- Notice in particular how \( \text{IFTHENELSE} \) simply applies the boolean to the two other arguments.

- Now we can define simple predicates
  - \( \text{ZERO?} := \lambda n. n \ (\lambda x. \text{FALSE}) \ \text{TRUE} \)
  - \( \text{LEQ} := \lambda mn. \text{ZERO?} (\text{SUB} m n) \)
  - \( \text{EQ} := \lambda mn. \text{AND} (\text{LEQ} m n) (\text{LEQ} n m) \)
The booleans operations are straight-forward.

- **AND**: \( \lambda p q. p \ q \ p \)
- **OR**: \( \lambda p q. p \ p \ q \)
- **NOT**: \( \lambda p a b. p \ b \ a \)
- **IFTHENELSE**: \( \lambda p a b. p \ a \ b \)

Notice in particular how **IFTHENELSE** simply applies the boolean to the two other arguments.

Now we can define simple predicates

- **ZERO?**: \( \lambda n. n \ (\lambda x. \text{FALSE}) \ \text{TRUE} \)
- **LEQ**: \( \lambda mn. \text{ZERO?} \ (\text{SUB} \ m \ n) \)
- **EQ**: \( \lambda mn. \text{AND} \ (\text{LEQ} \ m \ n) \ (\text{LEQ} \ n \ m) \)
Amazingly, our definition of booleans has the side-effect of allowing us to construct linked lists!

- PAIR := λxyf.f x y  
  construct a pair
- FIRST := λp.p TRUE  
  get the first element
- SECOND := λp.p FALSE  
  get the second element
- NIL := λx.TURE  
  the empty list
- NULL? := λp.p (λxy.FALSE)  
  test for the empty list

Lists are defined as NIL or a PAIR of an item and a list.
Pairs can be useful in their own right.

We define $\Phi: (m, n) \mapsto (n, n + 1)$ as

$\Phi := \lambda x. \text{PAIR} \ (\text{SECOND} \ x) \ (\text{SUCC} \ (\text{SECOND} \ x))$

Then $\text{PRED} := \lambda n. \text{FIRST} \ (n \ \Phi \ \text{PAIR} \ 0 \ 0)$

Finally $\text{SUB} := \lambda mn. n \ \text{PRED} \ m$
Pairs can be useful in their own right.

We define $\Phi : (m, n) \mapsto (n, n + 1)$ as
\[
\Phi := \lambda x. \text{PAIR} \ (\text{SECOND} \ x) \ (\text{SUCC} \ (\text{SECOND} \ x))
\]

Then
\[
PRED := \lambda n. \text{FIRST} \ (n \ \Phi \ (\text{PAIR} \ 0 \ 0))
\]

Finally
\[
SUB := \lambda mn. n \ PRED \ m
\]
Now we can use what we have learnt to write a programming that calculates Fibonacci numbers!

Let $\Phi := \lambda x.\text{PAIR} \ (\text{SECOND} \ x) \ (\text{PLUS} \ (\text{FIRST} \ x) \ (\text{SECOND} \ x))$

Then $\text{FIB} := \lambda n.\text{FIRST} \ (n \ \Phi \ (\text{PAIR} \ 0 \ 1))$

Or written in full:

$$
\lambda n. (n \ (\lambda p. (\lambda xyf. f \ x \ y) \ (p \ \lambda xy.y)) \\
(\lambda mnfx. m \ f \ (n \ f \ x)) \ (p \ \lambda xy.x) \ (p \ \lambda xy.y)) \\
(\lambda xyf. f \ x \ y) \ (\lambda fx. x) \ \lambda fx.f \ x) \ \lambda xy.x
$$
Example – Fibonacci Numbers

Now we can use what we have learnt to write a programming that calculates Fibonacci numbers!

Let \( \Phi := \lambda x. \text{PAIR} \ \text{SECOND} \ x \ \text{PLUS} \ \text{FIRST} \ x \ \text{SECOND} \ x \) 

Then \( \text{FIB} := \lambda n. \text{FIRST} \ (n \ \Phi \ (\text{PAIR} \ 0 \ 1)) \)

Or written in full:

\[
\lambda n. (n \ (\lambda p. (\lambda x y f. f \ x \ y) \ (p \ \lambda x y. y))
\ (\lambda m n f x. m \ f \ (n \ f \ x)) \ (p \ \lambda x y. x) \ (p \ \lambda x y. y))
\ (\lambda x y f. f \ x \ y) \ (\lambda f x. x) \ \lambda f x. f \ x) \ \lambda x y. x
\]
Example – Fibonacci Numbers

Now we can use what we have learnt to write a programming that calculates Fibonacci numbers!

Let $\Phi := \lambda x. \text{PAIR (SECOND x) (PLUS (FIRST x) (SECOND x))}$

Then $\text{FIB} := \lambda n. \text{FIRST (n \Phi (PAIR 0 1))}$

Or written in full:

$$\lambda n. (n \ (\lambda p. (\lambda xyf. f x y) \ (p \ \lambda xy. y)) \ ((\lambda mnf. m f (n f x)) \ (p \ \lambda xy. x) \ (p \ \lambda xy. y))) \ ((\lambda xyf. f x y) \ (\lambda fx. x) \ \lambda fx. f x)) \ \lambda xy. x$$
Example – Fibonacci Numbers

Now we can use what we have learnt to write a programming that calculates Fibonacci numbers!

Let $\Phi := \lambda x.\text{PAIR} \ (\text{SECOND} \ x) \ (\text{PLUS} \ (\text{FIRST} \ x) \ (\text{SECOND} \ x))$

Then $\text{FIB} := \lambda n.\text{FIRST} \ (n \ \Phi \ (\text{PAIR} \ 0 \ 1))$

Or written in full:

$\lambda n. (n \ (\lambda p. \ (\lambda xyf. f \ x \ y) \ (p \ \lambda xy. y) \ ((\lambda mnfx. m \ f \ (n \ f \ x)) \ (p \ \lambda xy. x) \ (p \ \lambda xy. y)))
((\lambda yf. f \ x \ y) \ (\lambda fx. x) \ \lambda fx. f \ x)) \ \lambda xy. x$
We already know how to do a lot in lambda calculus, but there are something we cannot do without recursion.

Well that’s fine. Let’s implement factorials recursively:

\[
FAC := \lambda n. \text{IFTHENELSE} \ (\text{LEQ} \ n \ 1) \ 1 \ (\text{MULT} \ n \ (FAC \ (\text{PRED} \ n)))
\]

Nope. Sorry. ‘\(FAC :=\)’ is just syntactic sugar that lets us build large terms more easily.

We need to be able to expand the result using only anonymous functions. Thus, we cannot call \(FAC\) inside the definition.
We already know how to do a lot in lambda calculus, but there are something we cannot do without recursion.

Well that’s fine. Let’s implement factorials recursively:

\[
\text{FAC} := \lambda n.\text{IFTHENELSE} (\text{LEQ} n 1) 1 (\text{MULT} n (\text{FAC} (\text{PRED} n)))
\]

Nope. Sorry. ‘FAC :=’ is just syntactic sugar that lets us build large terms more easily.

We need to be able to expand the result using only anonymous functions. Thus, we cannot call FAC inside the definition.
Recursion

- We already know how to do a lot in lambda calculus, but there are something we cannot do without recursion.
- Well that’s fine. Let’s implement factorials recursively:
  \[
  \text{FAC} := \lambda n. \text{IFTHENELSE} (\text{LEQ} n 1) 1 (\text{MULT} n (\text{FAC} (\text{PRED} n)))
  \]
- Nope. Sorry. ‘\text{FAC} :=’ is just syntactic sugar that lets us build large terms more easily.
- We need to be able to expand the result using only anonymous functions. Thus, we cannot call \text{FAC} inside the definition.
This is where we employ a trick: we pass $\text{FAC}$ another argument: a function that will be used to calculate the next iteration of the recursion.

$$\text{FAC} := \lambda n. \text{IFTHENELSE} (\text{LEQ} n 1) 1 (\text{MULT} n (f (\text{PRED} n)))$$

But wait, what function do we pass as the argument? Are we not back where we started?
This is where we employ a trick: we pass \( \text{FAC} \) another argument: a function that will be used to calculate the next iteration of the recursion.

\[
\text{FAC} := \lambda fn. \text{IFTHENELSE} (\text{LEQ} n 1) 1 (\text{MULT} n (f (\text{PRED} n)))
\]

But wait, what function do we pass as the argument? Are we not back where we started?
A Fixed-point Combinator

Let us assume for a moment that we could define a function $\text{FIX}$ with the curious property that

$$\text{FIX } f \equiv \beta f \ (\text{FIX } f)$$

Such a beast is called a fixed-point combinator as it finds the fixed-point of whatever function it is applied to it.
Anonymous Recursion

Returning to our problem

\[ \text{FAC} := \lambda fn. \text{IFTHENELSE} \left( \text{LEQ} \ n \ 1 \right) \ 1 \ \left( \text{MULT} \ n \ (f \ (\text{PRED} \ n)) \right) \]

What happens if we evaluate \( \text{FIX} \ \text{FAC} \)?

\[ \text{FIX} \ \text{FAC} \equiv_{\beta} \text{FAC} \ (\text{FIX} \ \text{FAC}) \]

Thus \( \text{FIX} \ \text{FAC} \) passes to \( \text{FAC} \) the exact function that is needed to calculate the factorial on the next step of the iteration!

\( \text{FIX} \ \text{FAC} \) is the factorial function we sought.
Now, that’s all well and good, but how can we be sure that such a function as `FIX` exists?

Simple, I can construct one:

\[ Y := \lambda f. (\lambda x.f (x x)) (\lambda x.f (x x)) \]

Let’s check that it indeed has the right properties:

\[
Y \ g = (\lambda f. (\lambda x.f (x x)) (\lambda x.f (x x))) \ g \\
\equiv_\beta (\lambda x.g (x x)) (\lambda x.g (x x)) \\
\equiv_\alpha (\lambda y.g (y y)) (\lambda x.g (x x)) \\
\equiv_\beta g ((\lambda x.g (x x)) (\lambda x.g (x x))) \\
= g (Y \ g)
\]
Now, that’s all well and good, but how can we be sure that such a function as $\text{FIX}$ exists?

Simple, I can construct one:

$$Y := \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

Let’s check that it indeed has the right properties:

$$Y \ g = (\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) \ g$$

$$\equiv_\beta (\lambda x. g (x x)) (\lambda x. g (x x))$$

$$\equiv_\alpha (\lambda y. g (y y)) (\lambda x. g (x x))$$

$$\equiv_\beta g ((\lambda x. g (x x)) (\lambda x. g (x x)))$$

$$= g (Y \ g)$$
Now, that’s all well and good, but how can we be sure that such a function as $\text{FIX}$ exists?

Simple, I can construct one:

$$Y := \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

Let’s check that it indeed has the right properties:

$$Y \; g = (\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) \; g$$

$$\equiv_{\beta} (\lambda x. g (x x)) (\lambda x. g (x x))$$

$$\equiv_{\alpha} (\lambda y. g (y y)) (\lambda x. g (x x))$$

$$\equiv_{\beta} g ((\lambda x. g (x x)) (\lambda x. g (x x)))$$

$$= g \; (Y \; g)$$
Now, that’s all well and good, but how can we be sure that such a function as \texttt{FIX} exists?

Simple, I can construct one:

\[
Y := \lambda f. (\lambda x. f \ (x \ x)) \ (\lambda x. f \ (x \ x))
\]

Let’s check that it indeed has the right properties:

\[
Y \ g = (\lambda f. (\lambda x. f \ (x \ x)) \ (\lambda x. f \ (x \ x))) \ g \\
\equiv_{\beta} (\lambda x. g \ (x \ x)) \ (\lambda x. g \ (x \ x)) \\
\equiv_{\alpha} (\lambda y. g \ (y \ y)) \ (\lambda x. g \ (x \ x)) \\
\equiv_{\beta} g \ ((\lambda x. g \ (x \ x)) \ (\lambda x. g \ (x \ x))) \\
= g \ (Y \ g)
\]
Now, that’s all well and good, but how can we be sure that such a function as $\text{FIX}$ exists?

Simple, I can construct one:

$$Y := \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

Let’s check that it indeed has the right properties:

$$Y \ g = (\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) \ g$$
$$\equiv_\beta (\lambda x. g (x x)) (\lambda x. g (x x))$$
$$\equiv_\alpha (\lambda y. g (y y)) (\lambda x. g (x x))$$
$$\equiv_\beta g ((\lambda x. g (x x)) (\lambda x. g (x x)))$$
$$= g (Y \ g)$$
Now, that’s all well and good, but how can we be sure that such a function as $\text{FIX}$ exists?

Simple, I can construct one:

$$Y := \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

Let’s check that it indeed has the right properties:

$$Y \ g = (\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) \ g$$

$$\equiv_\beta (\lambda x. g (x x)) (\lambda x. g (x x))$$

$$\equiv_\alpha (\lambda y. g (y y)) (\lambda x. g (x x))$$

$$\equiv_\beta g ((\lambda x. g (x x)) (\lambda x. g (x x)))$$

$$= g (Y \ g)$$
Now, that’s all well and good, but how can we be sure that such a function as \( \text{FIX} \) exists?

Simple, I can construct one:

\[
Y := \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))
\]

Let’s check that it indeed has the right properties:

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

\[
\equiv_\beta (\lambda x. g (x x)) (\lambda x. g (x x))
\]

\[
\equiv_\alpha (\lambda y. g (y y)) (\lambda x. g (x x))
\]

\[
\equiv_\beta g ((\lambda x. g (x x)) (\lambda x. g (x x)))
\]

\[
= g (Y \ g)
\]
Now, that’s all well and good, but how can we be sure that such a function as $\text{FIX}$ exists?

Simple, I can construct one:

$$Y := \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

Let’s check that it indeed has the right properties:

$$Y \ g = (\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) \ g$$

$$\equiv_\beta (\lambda x. g (x x)) (\lambda x. g (x x))$$

$$\equiv_\alpha (\lambda y. g (y y)) (\lambda x. g (x x))$$

$$\equiv_\beta g ((\lambda x. g (x x)) (\lambda x. g (x x)))$$

$$= g (Y \ g)$$
Outline

1. Introduction to Lambda Calculus
2. Untyped Lambda Calculus in Action
3. Typed Lambda Calculi
Untyped lambda calculus is interesting as a model of computation (in addition to being totally awesome)

But it is the typed calculi that are the most influential in modern language design.

Every expression is a typed lambda calculus has a well-defined type which restricts what operations can be performed on it.

This is analogous to types in programming languages with strong static typing.
Simply typed lambda calculus ($\lambda \rightarrow$) is lambda with a single type constructor $\rightarrow$ which constructs function types.

A type $\tau$ can either belong to a set of base types $B$ or it can be constructed from other types using $\rightarrow$.

- $\tau \in B$
- $\tau = \sigma \rightarrow \rho$

We must define a set of term constants for each base type.

For example, if we use the base type $nat$, our term constants will be the natural numbers.

It often turns out to be sufficient to consider only one base type $\sigma$ that has no term constants.
Simply Typed Lambda Calculus

- Simply typed lambda calculus \((\lambda \rightarrow)\) is lambda with a single type constructor \(\rightarrow\) which constructs function types.
- A type \(\tau\) can either belong to a set of base types \(B\) or it can be constructed from other types using \(\rightarrow\).
  - \(\tau \in B\)
  - \(\tau = \sigma \rightarrow \rho\)
- We must define a set of term constants for each base type.
- For example, if we use the base type \(\text{nat}\), our term constants will be the natural numbers.
- It often turns out to be sufficient to consider only one base type \(o\) that has no term constants.
The syntactic rules from the untyped calculus still apply.
Now term constants may be lambda terms.
A colon is used to assign a type to variables when they are bound. \( \lambda x : \tau . \lambda y : \sigma . f \ x \ y \)
Typed lambda calculus imposes the additional rule that expressions must ‘type check’.

If \( x \) is bound with type \( \tau \), it has type \( \tau \) in the scope of the abstraction.

Term constants are of the appropriate base type.

If \( e \) has type \( \tau \), then \( \lambda x: \sigma . e \) has type \( \sigma \rightarrow \tau \)

If \( f \) has type \( \sigma \rightarrow \tau \) and \( e \) has type \( \sigma \) then \( f \ e \) has type \( \tau \).
Consequences

- The extra restrictions imposed by simply typed lambda calculus now make $\beta$-reduction strongly normalising. That is, every expression has a normal form.

- This simplifies the analysis but implies simply typed calculus is no longer Turing complete.

- In particular, there is no way to define a function $\textbf{FIX}_\tau$ with type $(\tau \to \tau) \to \tau$.

- If this is added artificially the calculus becomes Turning complete once more.

- Even without doing this, all of the examples we covered before fixed-point combinators can still hold.
You may or may not have noticed that simply typed lambda calculus requires that we define an infinite number of identity functions – one for each type.

*System F* has a more powerful type system which can handle such *parametric polymorphism*.

It introduces *type variables* which can be bound to their own type of abstraction.
Examples

- We can define a polymorphic identity function
  \[ \Lambda \alpha. \lambda x^\alpha. x : \forall \alpha. \alpha \rightarrow \alpha \]

- If we let Boolean be the type \( \forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha \)
  \[ \text{TRUE} := \Lambda \alpha. \lambda x^\alpha. \lambda y^\alpha. x \]
  \[ \text{FALSE} := \Lambda \alpha. \lambda x^\alpha. \lambda y^\alpha. y \]

And then

- \( \text{AND} := \lambda x^{\text{Boolean}}. \lambda y^{\text{Boolean}}. x \text{ Boolean } y \text{ FALSE } \)
- \( \text{OR} := \lambda x^{\text{Boolean}}. \lambda y^{\text{Boolean}}. x \text{ Boolean } \text{ TRUE } y \)
- \( \text{NOT} := \lambda x^{\text{Boolean}}. x \text{ Boolean } \text{ FALSE } \text{ TRUE } \)
- \( \text{IFTHENELSE} := \Lambda \alpha. \lambda x^{\text{Boolean}}. \lambda y^\alpha. \lambda z^\alpha. x^\alpha y z \)
- \( \text{ISZER0} := \lambda n^{\forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha}. n \text{ Boolean } (\lambda x^{\text{Boolean}}. \text{ FALSE }) \text{ TRUE } \)
Examples

- We can define a polymorphic identity function
  \[ \Lambda \alpha. \lambda x^\alpha . x : \forall \alpha. \alpha \rightarrow \alpha \]

- If we let Boolean be the type \[ \forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha \]
  - TRUE := \[ \Lambda \alpha. \lambda x^\alpha . \lambda y^\alpha . x \]
  - FALSE := \[ \Lambda \alpha. \lambda x^\alpha . \lambda y^\alpha . y \]

And then
  - AND := \lambda x^{\text{Boolean}} . \lambda y^{\text{Boolean}} . x \ \text{Boolean} \ y \ \text{FALSE}
  - OR := \lambda x^{\text{Boolean}} . \lambda y^{\text{Boolean}} . x \ \text{Boolean} \ \text{TRUE} \ y
  - NOT := \lambda x^{\text{Boolean}} . x \ \text{Boolean} \ \text{FALSE} \ \text{TRUE}
  - IFTHENELSE := \[ \Lambda \alpha. \lambda x^{\text{Boolean}} . \lambda y^\alpha . \lambda z^\alpha . x^\alpha y z \]

- ISZERO :=
  \[ \lambda n^{\forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha} . n \ \text{Boolean} \ (\lambda x^{\text{Boolean}}. \text{FALSE}) \ \text{TRUE} \]
Examples

- We can define a polymorphic identity function
  \[ \Lambda \alpha. \lambda x^\alpha. x : \forall \alpha. \alpha \to \alpha \]
- If we let Boolean be the type \( \forall \alpha. \alpha \to \alpha \to \alpha \)
  - TRUE := \( \Lambda \alpha. \lambda x^\alpha. \lambda y^\alpha. x \)
  - FALSE := \( \Lambda \alpha. \lambda x^\alpha. \lambda y^\alpha. y \)
  And then
    - AND := \( \lambda x^{\text{Boolean}}. \lambda y^{\text{Boolean}}. x \text{ Boolean } y \text{ FALSE } \)
    - OR := \( \lambda x^{\text{Boolean}}. \lambda y^{\text{Boolean}}. x \text{ Boolean } \text{ TRUE } y \)
    - NOT := \( \lambda x^{\text{Boolean}}. x \text{ Boolean } \text{ FALSE } \text{ TRUE } \)
    - IFTHENELSE := \( \Lambda \alpha. \lambda x^{\text{Boolean}}. \lambda y^\alpha. \lambda z^\alpha. x_{\alpha yz} \)
    - ISZERO := \( \lambda n^{\forall \alpha. (\alpha \to \alpha) \to \alpha \to \alpha}. n \text{ Boolean } (\lambda x^{\text{Boolean}}. \text{ FALSE }) \text{ TRUE } \)
Examples

- We can define a polymorphic identity function
  \[ \Lambda \alpha. \lambda x^\alpha. x : \forall \alpha. \alpha \rightarrow \alpha \]

- If we let Boolean be the type \[ \forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha \]
  - \text{TRUE} := \Lambda \alpha. \lambda x^\alpha. \lambda y^\alpha. x
  - \text{FALSE} := \Lambda \alpha. \lambda x^\alpha. \lambda y^\alpha. y

And then

- \text{AND} := \lambda x^{\text{Boolean}}. \lambda y^{\text{Boolean}}. x \text{ Boolean } y \text{ FALSE}
- \text{OR} := \lambda x^{\text{Boolean}}. \lambda y^{\text{Boolean}}. x \text{ Boolean } \text{ TRUE} y
- \text{NOT} := \lambda x^{\text{Boolean}}. x \text{ Boolean } \text{ FALSE} \text{ TRUE}
- \text{IFTHENELSE} := \Lambda \alpha. \lambda x^{\text{Boolean}}. \lambda y^\alpha. \lambda z^\alpha. x^\alpha y z

- \text{ISZERO} := \\
  \lambda n^{\forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha}. n \text{ Boolean } (\lambda x^{\text{Boolean}}. \text{ FALSE}) \text{ TRUE}
System F allows us to encode arbitrary recursive abstract structures.

For example take the type of the Natural Numbers $\mathcal{N}$

- $\text{zero} : \mathcal{N}$
- $\text{succ} : \mathcal{N} \rightarrow \mathcal{N}$

We see that given a successor function $\text{succ}$ and $\text{zero}$ we can define a number $n$ by $n := \text{succ}^n(\text{zero})$

Thus, we construct a function that takes something of type $\mathcal{N} \rightarrow \mathcal{N}$ and of type $\mathcal{N}$ and returns resulting number with these as the definitions of $\text{succ}$ and $\text{zero}$.

Don’t know $\mathcal{N}$ so we make it polymorphic over all types.
System F allows us to encode arbitrary recursive abstract structures.

For example take the type of the Natural Numbers $\mathcal{N}$

- $\text{zero} : \mathcal{N}$
- $\text{succ} : \mathcal{N} \to \mathcal{N}$

We see that given a successor function $\text{succ}$ and $\text{zero}$ we can define a number $n$ by $n := \text{succ}^{(n)}(\text{zero})$

Thus, we construct a function that takes something of type $\mathcal{N} \to \mathcal{N}$ and of type $\mathcal{N}$ and returns resulting number with these as the definitions of $\text{succ}$ and $\text{zero}$.

Don’t know $\mathcal{N}$ so we make it polymorphic over all types.
System F allows us to encode arbitrary recursive abstract structures.

For example take the type of the Natural Numbers $\mathcal{N}$
- $\text{zero} : \mathcal{N}$
- $\text{succ} : \mathcal{N} \rightarrow \mathcal{N}$

We see that given a successor function $\text{succ}$ and $\text{zero}$ we can define a number $n$ by $n := \text{succ}^n(\text{zero})$

Thus, we construct a function that takes something of type $\mathcal{N} \rightarrow \mathcal{N}$ and of type $\mathcal{N}$ and returns resulting number with these as the definitions of $\text{succ}$ and $\text{zero}$.

Don’t know $\mathcal{N}$ so we make it polymorphic over all types.
System F allows us to encode arbitrary recursive abstract structures.

For example take the type of the Natural Numbers $\mathbb{N}$

- zero : $\mathbb{N}$
- succ : $\mathbb{N} \rightarrow \mathbb{N}$

We see that given a successor function $\text{succ}$ and zero we can define a number $n$ by $n := \text{succ}^{(n)}(\text{zero})$

Thus, we construct a function that takes something of type $\mathbb{N} \rightarrow \mathbb{N}$ and of type $\mathbb{N}$ and returns resulting number with these as the definitions of $\text{succ}$ and zero.

Don’t know $\mathbb{N}$ so we make it polymorphic over all types.
Thus we have,

- $0 := \Lambda \alpha. \lambda f^{\alpha \rightarrow \alpha} . \lambda x^{\alpha} . x$
- $1 := \Lambda \alpha. \lambda f^{\alpha \rightarrow \alpha} . \lambda x^{\alpha} . f x$
- $2 := \Lambda \alpha. \lambda f^{\alpha \rightarrow \alpha} . \lambda x^{\alpha} . f(f x)$
- etc.

These correspond the original definition of Church numerals!
Uses of System F

- System F is still strongly normalising.
- It comes close to representing the powerful type systems of languages such as Haskell.
- It provides a simple, well-defined system for dealing with types in these languages.
- System F can be further extended to System F< which including subtyping.
- This system is very important in language design as it models the type systems of many real languages.
System F and similar systems are useful for type inference where a compiler determines the type of an expression without the programmer having to specify it explicitly.

In 1994, Joe Wells proved that type inference was undecidable in System F.

However, a slightly restricted version called Hindley-Milner has fairly simple algorithm which was used in many languages today.
Untyped lambda calculus has had a wide-ranging influence on many programming languages including Lisp and ALGOL which went on to affect hundreds of other languages.

Lambda Calculus influenced the use of lexical scope, first class functions and lead to the consideration of lazy versus eager evaluation.

Howard-Curry correspondence means that lambda calculus is used heavily in automated theorem provers.