The Lambda Calculus

The lambda-calculus

- If our previous language of arithmetic expressions was the simplest nontrivial programming language, then the lambda-calculus is the simplest interesting programming language...
  - Turing complete
  - higher order (functions as data)
- Indeed, in the lambda-calculus, all computation happens by means of function abstraction and application.
- The *e. coli* of programming language research
- The foundation of many real-world programming language designs (including ML, Haskell, Scheme, Lisp, ...)

Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write

\[ \text{plus3 } x = \text{succ (succ (succ x))} \]

That is, “\text{plus3 } x \text{ is succ (succ (succ x))}.”

Q: What is \text{plus3} itself?

A: \text{plus3} is the function that, given \( x \), yields \( \text{succ (succ (succ x))} \).
Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write

\[ \text{plus3 } x = \text{succ (succ (succ x))} \]

That is, \( \text{plus3 } x \text{ is succ (succ (succ x))} \).

Q: What is \text{plus3} itself?
A: \text{plus3} is the function that, given \( x \), yields \( \text{succ (succ (succ x))} \).

\[ \text{plus3} = \lambda x. \text{succ (succ (succ x))} \]

This function exists independent of the name \text{plus3}.

\( \lambda x. \ t \) is written “\text{fun x } \rightarrow \ t\” in OCaml.

Abstractions over Functions

Consider the \( \lambda \)-abstraction

\[ g = \lambda f. f (f (\text{succ 0})) \]

Note that the parameter variable \( f \) is used in the \text{function} position in the body of \( g \). Terms like \( g \) are called higher-order functions. If we apply \( g \) to an argument like \text{plus3}, the “substitution rule” yields a nontrivial computation:

\[ g \text{ plus3} = (\lambda f. f (f (\text{succ 0}))) (\lambda x. \text{succ (succ (succ x)))} \]

\( i.e. \)

\[ (\lambda x. \text{succ (succ (succ x)))} (\text{success 0}) \]

\( i.e. \)

\[ (\lambda x. \text{succ (succ (succ x)))} (\text{success 0}) \]

\( i.e. \)

\[ \text{succ (succ (succ (succ (succ 0))))} \]

Abstractions Returning Functions

Consider the following variant of \( g \):

\[ \text{double} = \lambda f. \lambda y. f (f y) \]

I.e., \( \text{double} \) is the function that, when applied to a function \( f \), yields a \text{function} that, when applied to an argument \( y \), yields \( f (f y) \).

Example

\[ \text{double plus3 0} = (\lambda f. \lambda y. f (f y)) \]

\[ (\lambda x. \text{succ (succ (succ x)))} 0 \]

\( i.e. \)

\[ (\lambda y. (\lambda x. \text{succ (succ (succ x)))} y)) 0 \]

\( i.e. \)

\[ (\lambda x. \text{succ (succ (succ x)))} (\text{success 0}) \]

\( i.e. \)

\[ \text{succ (succ (succ (succ (succ 0))))} \]

The Pure Lambda-Calculus

As the preceding examples suggest, once we have \( \lambda \)-abstraction and application, we can throw away all the other language primitives and still have left a rich and powerful programming language.

In this language — the “pure lambda-calculus” — \text{everything} is a function.

▶ Variables always denote functions
▶ Functions always take other functions as parameters
▶ The result of a function is always a function
Formalities

Syntax

\[ t ::= \]
- terms
  - \( x \) variable
  - \( \lambda x.t \) abstraction
  - \( t \) application

Terminology:
- terms in the pure \( \lambda \)-calculus are often called \( \lambda \)-terms
- terms of the form \( \lambda x. t \) are called \( \lambda \)-abstractions or just abstractions

Syntactic conventions

Since \( \lambda \)-calculus provides only one-argument functions, all multi-argument functions must be written in curried style.

The following conventions make the linear forms of terms easier to read and write:

- Application associates to the left
  
  \[ E.g., t \ u \ v \text{ means } (t \ u) \ v, \text{ not } t \ (u \ v) \]

- Bodies of \( \lambda \)-abstractions extend as far to the right as possible

  \[ E.g., \lambda x. \lambda y. x \ y \text{ means } \lambda x. (\lambda y. x \ y), \text{ not } \lambda x. (\lambda y. x) \ y \]

Scope

The \( \lambda \)-abstraction term \( \lambda x. t \) binds the variable \( x \).

The scope of this binding is the body \( t \).

Occurrences of \( x \) inside \( t \) are said to be \textit{bound} by the abstraction.

Occurrences of \( x \) that are not within the scope of an abstraction binding \( x \) are said to be \textit{free}.

Values

\[ v ::= \]
- values
  - \( \lambda x.t \) abstraction value

\[ \lambda x. x \ y \ z \]

\[ \lambda x. (\lambda y. x \ y) \ y \]
**Operational Semantics**

Computation rule:

\[(\lambda x.t_{12}) v_2 \rightarrow [x \mapsto v_2]t_{12} \quad (E\text{-AppAbs})\]

*Notation: \([x \mapsto v_2]t_{12}\) is "the term that results from substituting free occurrences of \(x\) in \(t_{12}\) with \(v_{12}\)."

**Congruence rules:**

\[t_1 \rightarrow t_1' \quad (E\text{-App1})\]
\[t_1 t_2 \rightarrow t_1' t_2 \quad (E\text{-App2})\]

**Terminology**

A term of the form \((\lambda x. t) v\) — that is, a \(\lambda\)-abstraction applied to a value — is called a redex (short for "reducible expression").

**Alternative evaluation strategies**

Strictly speaking, the language we have defined is called the pure, call-by-value lambda-calculus. The evaluation strategy we have chosen — call by value — reflects standard conventions found in most mainstream languages. Some other common ones:

- Call by name (cf. Haskell)
- Normal order (leftmost/outermost)
- Full (non-deterministic) beta-reduction

**Multiple arguments**

Above, we wrote a function `double` that returns a function as an argument.

\[double = \lambda f. \lambda y. f (f y)\]

This idiom — a \(\lambda\)-abstraction that does nothing but immediately yield another abstraction — is very common in the \(\lambda\)-calculus. In general, \(\lambda x. \lambda y. t\) is a function that, given a value \(v\) for \(x\), yields a function that, given a value \(u\) for \(y\), yields \(t\) with \(v\) in place of \(x\) and \(u\) in place of \(y\). That is, \(\lambda x. \lambda y. t\) is a two-argument function.

(Recall the discussion of currying in OCaml.)
Syntactic conventions

Since $\lambda$-calculus provides only one-argument functions, all multi-argument functions must be written in curried style. The following conventions make the linear forms of terms easier to read and write:

- Application associates to the left
  
  E.g., $t \ u \ v$ means $(t \ u) \ v$, not $t \ (u \ v)$

- Bodies of $\lambda$-abstractions extend as far to the right as possible
  
  E.g., $\lambda x. \lambda y. \ x \ y$ means $\lambda x. (\lambda y. \ x \ y)$, not $\lambda x. (\lambda y. \ x) \ y$

---

The “Church Booleans”

\[
\begin{align*}
\text{tru} &= \lambda t. \lambda f. t \\
\text{fls} &= \lambda t. \lambda f. f
\end{align*}
\]

That is, \text{not} is a function that, given a boolean value $v$, returns \text{fls} if $v$ is \text{tru} and \text{tru} if $v$ is \text{fls}.

---

Functions on Booleans

\[
\begin{align*}
\text{not} &= \lambda b. \ b \ \text{fls} \ \text{tru}
\end{align*}
\]

That is, \text{not} is a function that, given a boolean value $v$, returns \text{fls} if $v$ is \text{tru} and \text{tru} if $v$ is \text{fls}.

---

Pairs

\[
\begin{align*}
pair &= \lambda f. \lambda s. \lambda b. \ b \ f \ s \\
\text{fst} &= \lambda p. \ p \ \text{tru} \\
\text{snd} &= \lambda p. \ p \ \text{fls}
\end{align*}
\]

That is, \text{pair} $v$ \ $w$ is a function that, when applied to a boolean value $b$, applies $b$ to $v$ and $w$.

By the definition of booleans, this application yields $v$ if $b$ is \text{tru} and $w$ if $b$ is \text{fls}, so the first and second projection functions \text{fst} and \text{snd} can be implemented simply by supplying the appropriate boolean.

---

Example

\[
\begin{align*}
fst \ (pair \ v \ w) &= \text{fst} \ ((\lambda f. \lambda s. \lambda b. \ b \ f \ s) \ v \ w) \ \text{by definition} \\
&\rightarrow \text{fst} \ ((\lambda s. \lambda b. \ b \ v \ s) \ v \ w) \ \text{reducing} \\
&\rightarrow \text{fst} \ (\lambda b. \ b \ v \ w) \ \text{reducing} \\
&= (\lambda p. \ p \ \text{tru}) \ (\lambda b. \ b \ v \ w) \ \text{by definition} \\
&\rightarrow (\lambda b. \ b \ v \ w) \ \text{tru} \ \text{reducing} \\
&\rightarrow \text{tru} \ v \ w \\
&\rightarrow^* v
\end{align*}
\]
Church numerals

Idea: represent the number \( n \) by a function that "repeats some action \( n \) times."

\[
\begin{align*}
c_0 &= \lambda s. \lambda z. z \\
c_1 &= \lambda s. \lambda z. s \ z \\
c_2 &= \lambda s. \lambda z. s \ (s \ z) \\
c_3 &= \lambda s. \lambda z. s \ (s \ (s \ z))
\end{align*}
\]

That is, each number \( n \) is represented by a term \( c_n \) that takes two arguments, \( s \) and \( z \) (for "successor" and "zero"), and applies \( s \), \( n \) times, to \( z \).

Functions on Church Numerals

Successor:

\[
scc = \lambda n. \lambda s. \lambda z. s \ (n \ s \ z)
\]

Addition:

\[
plus = \lambda m. \lambda n. \lambda s. \lambda z. m \ s \ (n \ s \ z)
\]

Multiplication:

\[
times = \lambda m. \lambda n. m \ (plus \ n) \ c_0
\]

Zero test:

\[
iszro = \lambda m. m \ (\lambda x. \text{fls}) \ \text{tru}
\]

What about predecessor?
Functions on Church Numerals

Successor:
\[ \text{succ} = \lambda n. \lambda s. \lambda z. s (n s z) \]

Addition:
\[ \text{plus} = \lambda m. \lambda n. \lambda s. \lambda z. m s (n s z) \]

Multiplication:
\[ \text{times} = \lambda m. \lambda n. m \ (\text{plus} \ n) \ c_0 \]

Zero test:
\[ \text{iszro} = \lambda m. m \ (\lambda x. \text{fls}) \ tru \]

What about predecessor?

Predecessor

\[ \text{zz} = \text{pair} \ c_0 \ c_0 \]
\[ \text{ss} = \lambda p. \text{pair} \ (\text{snd} \ p) \ (\text{succ} \ (\text{snd} \ p)) \]
\[ \text{prd} = \lambda m. \text{fst} \ (m \ ss \ \text{zz}) \]

Normal forms

Recall:
- A normal form is a term that cannot take an evaluation step.
- A stuck term is a normal form that is not a value.

Are there any stuck terms in the pure \( \lambda \)-calculus?
Prove it.
Normal forms

Recall:

- A normal form is a term that cannot take an evaluation step.
- A stuck term is a normal form that is not a value.

Are there any stuck terms in the pure λ-calculus? Prove it.
Does every term evaluate to a normal form? Prove it.

Divergence

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

Note that \( \omega \) evaluates in one step to itself!
So evaluation of \( \omega \) never reaches a normal form: it diverges.

Recursion in the Lambda-Calculus

Iterated Application

Suppose \( f \) is some \( \lambda \)-abstraction, and consider the following term:

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

Now the “pattern of divergence” becomes more interesting:

\[ Y_f =
\begin{align*}
(\lambda x. f (x x)) (\lambda x. f (x x)) \\
\rightarrow
f ((\lambda x. f (x x)) (\lambda x. f (x x))) \\
\rightarrow
f (f ((\lambda x. f (x x)) (\lambda x. f (x x)))) \\
\rightarrow
f (f (f ((\lambda x. f (x x)) (\lambda x. f (x x))))) \\
\rightarrow
\ldots
\end{align*} \]
Y_f is still not very useful, since (like omega), all it does is diverge. Is there any way we could "slow it down"?

Delaying divergence

\[ \text{poisonpill} = \lambda y. \omega \]

Note that poisonpill is a value — it will only diverge when we actually apply it to an argument. This means that we can safely pass it as an argument to other functions, return it as a result from functions, etc.

\[ (\lambda p. \text{fst} (\text{pair} p \text{fls}) \text{tru}) \text{poisonpill} \]
\[ \text{□} \rightarrow \text{fst} (\text{pair} \text{poisonpill fls}) \text{tru} \]
\[ \text{□} \rightarrow \ast \text{poisonpill tru} \]
\[ \text{□} \rightarrow \omega \]
\[ \text{□} \rightarrow \cdots \]

Cf. thunks in OCaml.

A delayed variant of omega

Here is a variant of omega in which the delay and divergence are a bit more tightly intertwined:

\[ \text{omegav} = \lambda y. (\lambda x. (\lambda y. x x y)) (\lambda x. (\lambda y. x x y)) y \]

Note that omegav is a normal form. However, if we apply it to any argument \( v \), it diverges:

\[ \text{omegav} v \]
\[ = (\lambda y. (\lambda x. (\lambda y. x x y)) (\lambda x. (\lambda y. x x y)) y) v \]
\[ = (\lambda x. (\lambda y. x x y)) (\lambda x. (\lambda y. x x y)) v \]
\[ = (\lambda y. (\lambda x. (\lambda y. x x y)) (\lambda x. (\lambda y. x x y)) y) v \]
\[ = \text{omegav} v \]

If we now apply Z_f to an argument v, something interesting happens:

\[ Z_f v \]
\[ = (\lambda y. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)) y) v \]
\[ \rightarrow (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)) v \]
\[ = f (\lambda y. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)) y) v \]
\[ = f Z_f v \]

Since Z_f and v are both values, the next computation step will be the reduction of f Z_f — that is, before we "diverge," f gets to do some computation.

Now we are getting somewhere.

Another delayed variant

Suppose \( f \) is a function. Define

\[ Z_f = \lambda y. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)) y \]

This term combines the "added f" from Y_f with the "delayed divergence" of omegav.

Recursion

Let

\[ f = \lambda fct. \]
\[ \lambda n. \]
\[ \text{if } n=0 \text{ then } 1 \]
\[ \text{else } n * (fct (\text{pred } n)) \]

f looks just the ordinary factorial function, except that, in place of a recursive call in the last time, it calls the function fct, which is passed as a parameter.

N.b.: for brevity, this example uses "real" numbers and booleans, infix syntax, etc. It can easily be translated into the pure lambda-calculus (using Church numerals, etc.).
We can use $Z$ to “tie the knot” in the definition of $f$ and obtain a real recursive factorial function:

$$Z_f\ 3
\xrightarrow{\cdot}\ f\ Z_f\ 3
=\ (\lambda fct.\ \lambda n.\ ...\ Z_f\ 3
\xrightarrow{\cdot}\ \text{if } 3=0\ \text{then } 1\ \text{else } 3 * (Z_f (\text{pred } 3))
\xrightarrow{\cdot}\ 3 * (Z_f (\text{pred } 3)))
\xrightarrow{\cdot}\ 3 * (Z_f\ 2)
\xrightarrow{\cdot}\ 3 * (f\ Z_f\ 2)
\cdot \cdot \cdot$$

For example:

$$\text{fact} = Z\ (\lambda fct.\ \lambda n.\ 
\text{if } n=0\ \text{then } 1
\text{else } n * (fct (\text{pred } n)))$$

A Generic $Z$

If we define

$$Z = \lambda f.\ Z_f$$

i.e.,

$$Z = \lambda f.\ \lambda y.\ (\lambda x.\ f (\lambda y.\ x x y))\ (\lambda x.\ f (\lambda y.\ x x y))\ y$$

then we can obtain the behavior of $Z_f$ for any $f$ we like, simply by applying $Z$ to $f$.

$$Z\ f \xrightarrow{\cdot}\ Z_f$$

Technical Note

The term $Z$ here is essentially the same as the $\text{fix}$ discussed the book.

$$Z = \lambda f.\ \lambda y.\ (\lambda x.\ f (\lambda y.\ x x y))\ (\lambda x.\ f (\lambda y.\ x x y))\ y$$

$$\text{fix} = \lambda f.\ (\lambda x.\ f (\lambda y.\ x x y))\ (\lambda x.\ f (\lambda y.\ x x y))$$

$Z$ is hopefully slightly easier to understand, since it has the property that $Z\ f\ v \xrightarrow{\cdot}\ f\ (Z\ f)\ v$, which $\text{fix}$ does not (quite) share.