The Pure Lambda Calculus

Syntax

\[ t ::= \]
\[ x \] variable
\[ \lambda x.t \] abstraction
\[ t \ t \] application

Values

\[ v ::= \]
\[ \lambda x.t \] abstraction value
**Operational Semantics**

**Computation rule:**

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

\([x \mapsto v_2]t_{12}\) is “the term that results from substituting occurrences of \(x\) in \(t_{12}\) with \(v_2\).”

---

**Operational Semantics**

**Computation rule:**

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

\([x \mapsto v_2]t_{12}\) is “the term that results from substituting occurrences of \(x\) in \(t_{12}\) with \(v_2\).”

**Congruence rules:**

\[
\begin{align*}
\frac{t_1 \rightarrow t'_1}{t_1 \, t_2 \rightarrow t'_1 \, t_2} & \quad (E\text{-APP1}) \\
\frac{t_2 \rightarrow t'_2}{v_1 \, t_2 \rightarrow v_1 \, t'_2} & \quad (E\text{-APP2})
\end{align*}
\]

---

**Terminology**

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

---

**Alternative evaluation strategies**

The evaluation strategy we have chosen — called call by value — reflects standard conventions found in most mainstream languages.

Some other common ones:

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

On Monday, we wrote a function double that returns a function as an argument.

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

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.

Aside: Currying

The transformation from a function taking a pair of arguments (in a language like OCaml that provides pairs) to a one-argument function returning another one-argument function is called currying.

It is considered good style in OCaml to define functions in curried style whenever possible.

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
- Bodies of \(\lambda\)-abstractions extend as far to the right as possible
The “Church Booleans”

\[ \text{tru} = \lambda t. t \]
\[ \text{fls} = \lambda f. f \]

\[
\begin{align*}
\text{tru} \ v \ w \\
&= (\lambda t. t) \ v \ w \quad \text{by definition} \\
&\rightarrow (\lambda f. f) \ v \ w \quad \text{reducing the underlined redex} \\
&\rightarrow v \quad \text{reducing the underlined redex} \\
\text{fls} \ v \ w \\
&= (\lambda f. f) \ v \ w \quad \text{by definition} \\
&\rightarrow (\lambda f. f) \ v \ w \quad \text{reducing the underlined redex} \\
&\rightarrow w \quad \text{reducing the underlined redex}
\end{align*}
\]

Functions on Booleans

\[ \text{not} = \lambda b. \text{fls} \ \text{tru} \]

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

\[ \text{and} = \lambda b. \lambda c. b \ c \ \text{fls} \]

That is, \( \text{and} \) is a function that, given two boolean values \( v \) and \( w \), returns \( v \) if \( v \) is \( \text{tru} \) and \( \text{fls} \) if \( v \) is \( \text{fls} \). Thus \( \text{and} \ v \ w \) yields \( \text{tru} \) if both \( v \) and \( w \) are \( \text{tru} \) and \( \text{fls} \) if either \( v \) or \( w \) is \( \text{fls} \).

Pairs

\[ \text{pair} = \lambda f. \lambda s. \lambda b. \ b \ f \ s \]
\[ \text{fst} = \lambda p. \ p \ \text{tru} \]
\[ \text{snd} = \lambda p. \ p \ \text{fls} \]

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**

\[
\begin{align*}
\text{fst (pair } v \, w) & \\
\quad \rightarrow \text{fst } (\lambda f. \lambda s. \lambda b. b \, f \, s) \, v \, w) & \quad \text{by definition} \\
\quad \rightarrow \text{fst } (\lambda s. \lambda b. b \, v \, s) \, w) & \quad \text{reducing the underlined redex} \\
\quad \rightarrow \text{fst } (\lambda b. b \, v \, w) & \quad \text{reducing the underlined redex} \\
\quad \rightarrow (\lambda p. \text{tru}) \, (\lambda b. b \, v \, w) & \quad \text{by definition} \\
\quad \rightarrow (\lambda b. b \, v \, w) \, \text{tru} & \quad \text{reducing the underlined redex} \\
\quad \rightarrow \text{tru } v \, w & \quad \text{reducing the underlined redex} \\
\quad \rightarrow^* v & \quad \text{as before.}
\end{align*}
\]

**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:**

\[
scc = \lambda n. \lambda s. \lambda z. s \, (n \, s \, z)
\]
Functions on Church Numerals

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

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

Multiplication:
\[ \text{times} = \lambda m \lambda n \ m \ (\text{plus} \ n) \ c_0 \]
Functions on Church Numerals

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

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

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

Zero test:
\[ \text{iszero} = \lambda m. \; (\lambda x. \; \text{false}) \; \text{tru} \]

---

What about predecessor?

Functions on Church Numerals

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

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

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

Zero test:
\[ \text{iszero} = \lambda m. \; (\lambda x. \; \text{false}) \; \text{tru} \]
Normal forms

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.

Divergence

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

Note that \text{omega} evaluates in one step to itself.
So evaluation of \text{omega} never reaches a normal form: it diverges.
Divergence

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

Note that \text{omega} evaluates in one step to itself.
So evaluation of \text{omega} never reaches a normal form: it diverges.

Being able to write a divergent computation does not seem very useful in itself. However, there are variants of \text{omega} that are very useful...

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:

\[ Y_f \]
\[ = (\lambda x. f (x x)) (\lambda x. f (x x)) \]
\[ \rightarrow f ((\lambda x. f (x x)) (\lambda x. f (x x))) \]
\[ \rightarrow f (f ((\lambda x. f (x x)) (\lambda x. f (x x)))) \]
\[ \rightarrow f (f (f ((\lambda x. f (x x)) (\lambda x. f (x x))))) \]
\[ \rightarrow ... \]
Delaying Divergence

poisonpill = \lambda y. \text{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.

\[
(\lambda p. \text{fst} \ (\text{pair} \ p \ \text{fls}) \ \text{tru}) \ \text{poisonpill} \\
\quad \quad \quad \quad \rightarrow \\
\quad \quad \quad \quad \text{fst} \ (\text{pair} \ \text{poisonpill} \ \text{fls}) \ \text{tru} \\
\quad \quad \quad \quad \quad \quad \quad \quad \rightarrow \\
\quad \quad \quad \quad \quad \quad \quad \quad \text{poisonpill} \ \text{tru} \\
\quad \quad \quad \quad \quad \quad \quad \quad \rightarrow \\
\quad \quad \quad \quad \quad \quad \quad \quad \text{omega} \\
\quad \quad \quad \quad \quad \quad \quad \quad \rightarrow \\
\quad \quad \quad \quad \quad \quad \quad \quad \ldots
\]

A delayed variant of omega

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

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

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

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

Another delayed variant

Suppose f is a function. Define

\[
\text{Z}_f = \lambda y. (\lambda x. f \ (\lambda y. x \ y)) \ (\lambda x. f \ (\lambda y. x \ y)) \ y
\]

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

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

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

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.
Recursion

Let

\[ f = \lambda n. \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (f \text{ (pred } n) ) & \text{else} \end{cases} \]

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

N.b.: for brevity, this example uses “real” numbers and booleans, infix syntax, etc...

A Generic Z

If we define

\[ Z = \lambda f. Z_f \]

e.g.,

\[ Z = \lambda f. \lambda y. (\lambda x. f (\lambda y. x \cdot y)) (\lambda x. f (\lambda y. x \cdot y)) y \]

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

\[ Z f \rightarrow Z_f \]

We can use \( Z \) to “tie the knot” in the definition of \( f \) and obtain a real recursive factorial function:

\[ \begin{align*} Z_3 & \rightarrow^* \\ f Z_3 & \rightarrow \\ (\lambda n. \ldots) Z_3 & \rightarrow \\ \text{if } 3 = 0 \text{ then } 1 \text{ else } 3 \cdot (Z_3 \text{ (pred } 3)) & \rightarrow \\ 3 \cdot (Z_3 \text{ (pred } 3)) & \rightarrow \\ 3 \cdot (Z_2) & \rightarrow^* \\ 3 \cdot (f Z_2) & \ldots \end{align*} \]

N.b.: The term \( Z \) here is essentially the same as the \( \text{fix} \) discussed the book.

\[ \begin{align*} Z & = \lambda f. \lambda y. (\lambda x. f (\lambda y. x \cdot y)) (\lambda x. f (\lambda y. x \cdot y)) y \\ \text{fix} & = \lambda f. (\lambda x. f (\lambda y. x \cdot y)) (\lambda x. f (\lambda y. x \cdot y)) \end{align*} \]

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