Lambda-Calculus

CSCI 2210

2023-11-06 — 2023-11-13

Language-Based Models of Computable Functions

So far, each time we introduced a machine-based model of computation,

we also introduced a corresponding language-based model:

- finite state automata — regular expressions,
- pushdown automata — context-free grammars,
- Turing machines — ???

With Turing machines we are spoiled for choice.

Your favorite programming language is probably *Turing complete*.

The Original Programming Language

We will study an extremely simple programming language,

which predates both electronic computers and Turing machines.

It is in some sense the *original* programming language.

The λ-calculus was introduced in the early 1930s by Alonzo Church
to study mathematical functions.

Lambda Terms

We start with a countably infinite collection of **variables**, V.

We typically write variables using letters like x, y, z, v_1, v_2, etc.

An expression or **term** of the λ-calculus is given by the following inductive definition:

variable: if $x \in \mathrm{V}$ then x is a term,

application: if M and N are terms then $(\mathrm{M}\,\mathrm{N})$ is a term,

abstraction: if x is a variable and M is a term then $(\lambda\, x\, .\, \mathrm{M})$ is a term.

The set of λ-calculus terms is called "Λ".

Notational Conventions

Every term has a unique parse tree with internal nodes `var`, `app`, and `abs`.

To simplify notation we observe the following conventions:

- Drop outermost parentheses, so:

$$\mathrm{M\,N} \;:=\; (\mathrm{M\,N})$$

- Application is left associative, so:

$$\mathrm{M\,N\,P} \;:=\; (\mathrm{M\,N})\,\mathrm{P}$$

- The body of an abstraction extends as far as syntactically possible, so:

$$\lambda x\,.\,\mathrm{M\,N} \;:=\; \lambda x\,.\,(\mathrm{M\,N})$$

- Successive abstractions can be contracted, so:

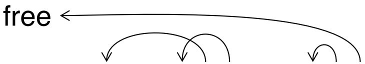
$$\lambda xyz\,.\,\mathrm{M} \;:=\; \lambda x\,.\,\lambda y\,.\,\lambda z\,.\,\mathrm{M}$$

Variable Binding

The "λ" in an abstraction **binds** occurrences of its variable within its scope.

Each non-binding occurrence of a variable is **bound** by the innermost abstraction of the same variable.

If a variable occurrence is not bound by any abstraction, then it is **free**.



$$\lambda x \,.\, (\lambda y \,.\, x\, y)(\lambda x \,.\, x\, y)$$

- The set of all variables occurring in term M is $\mathrm{var}(\mathrm{M})$.
- The set of free variables occurring in term M is $\mathrm{fv}(\mathrm{M})$.

Bound Variable Renaming

The **renaming** of variable x to variable y in term M, written "$\mathrm{M}\{y/x\}$", replaces all occurrences of x with y in M.

The purpose of bound variables is to specify the binding structure of terms. We don't actually care what bound variables are called.

So we adopt the following **inference rule**:

$$\frac{y \notin \mathrm{var}(\mathrm{M})}{\lambda x \,.\, \mathrm{M} \;=_\alpha\; \lambda y \,.\, \mathrm{M}\{y/x\}}\;\alpha$$

This says if y does not occur in M then the term $\lambda x \,.\, \mathrm{M}$ is α**-equivalent** to the term obtained by renaming x to y in M and binding the result with y instead of with x.

Alpha-Equivalence of Terms

The inference rule α induces an **equivalence relation** on terms:

$$\frac{}{\mathrm{M} =_\alpha \mathrm{M}}\ \alpha\text{-refl} \qquad \frac{\mathrm{M} =_\alpha \mathrm{N}}{\mathrm{N} =_\alpha \mathrm{M}}\ \alpha\text{-symm} \qquad \frac{\mathrm{M} =_\alpha \mathrm{N} \qquad \mathrm{N} =_\alpha \mathrm{P}}{\mathrm{M} =_\alpha \mathrm{P}}\ \alpha\text{-trans}$$

which is moreover a **congruence**, *compatible* with application and abstraction:

$$\frac{\mathrm{M} =_\alpha \mathrm{M}' \qquad \mathrm{N} =_\alpha \mathrm{N}'}{\mathrm{M}\,\mathrm{N} =_\alpha \mathrm{M}'\,\mathrm{N}'}\ \alpha\text{-app} \qquad \frac{\mathrm{M} =_\alpha \mathrm{N}}{\lambda x\,.\,\mathrm{M} =_\alpha \lambda x\,.\,\mathrm{N}}\ \alpha\text{-abs}$$

In λ-calculus we don't distinguish between α-equivalent terms and write "$=$" for $=_\alpha$.

Derivation Trees

We can combine inference rules into formal proofs called **derivation trees**:

$$\dfrac{\dfrac{\dfrac{}{y \notin \mathrm{var}(x)}}{\lambda x \,.\, x \;=_\alpha\; \lambda y \,.\, y}\;\alpha \qquad \dfrac{\dfrac{}{z \notin \mathrm{var}(y)}}{\lambda y \,.\, y \;=_\alpha\; \lambda z \,.\, z}\;\alpha}{\lambda x \,.\, x \;=_\alpha\; \lambda z \,.\, z}\;\alpha\text{-trans}$$

In order for such a tree to represent a completed proof, all of its leaves must be *closed*.

Substitution Desiderata

The most important operation in λ-calculus is the **substitution** of a term for a free variable in another term.

The notation "$\mathrm{M}[\mathrm{N}/x]$" represents the substitution of N for x in M.

When performing substitution we don't want to disturb the binding structure of terms:

- we don't want to substitute for **bound** variables in M,

  so $$(\lambda x \,.\, x)[y/x] \neq \lambda x \,.\, y$$

- we don't want to **capture** free variables in N,

  so $$(\lambda y \,.\, x)[y/x] \neq \lambda y \,.\, y$$

Substitution Definition

The **substitution** operation $\mathrm{M}[\mathrm{N}/x]$ is defined by recursion on the term M:

$$
\begin{array}{rcll}
x[\mathrm{N}/x] & = & \mathrm{N} & \\
y[\mathrm{N}/x] & = & y & \text{if } y \neq x \\
(\mathrm{M}\,\mathrm{P})[\mathrm{N}/x] & = & (\mathrm{M}[\mathrm{N}/x])\,(\mathrm{P}[\mathrm{N}/x]) & \\
(\lambda x\,.\,\mathrm{M})[\mathrm{N}/x] & = & \lambda x\,.\,\mathrm{M} & \\
(\lambda y\,.\,\mathrm{M})[\mathrm{N}/x] & = & \lambda y\,.\,(\mathrm{M}[\mathrm{N}/x]) & \text{if } x \neq y \text{ and } y \notin \mathrm{fv}(\mathrm{N}) \\
(\lambda y\,.\,\mathrm{M})[\mathrm{N}/x] & = & \lambda z\,.\,(\mathrm{M}\{z/y\}[\mathrm{N}/x]) & \text{if } x \neq y,\ y \in \mathrm{fv}(\mathrm{N}) \text{ and } z \text{ fresh}
\end{array}
$$

Beta-Reduction of Terms

An *application* of an *abstraction* to a term is called a β**-reducible expression**, or "β-redex":

$$\overbrace{(\underbrace{\lambda x \,.\, M}_{\text{abstraction}})\, N}^{\text{application}}$$

The idea is that of applying a function to an argument.

The operation of β**-reduction** relates a β-redex to the term obtained by substituting the argument for the bound variable in the body of the function. (Just like calling a function in programming!)

$$\underbrace{(\lambda x \,.\, M)\, N}_{\text{redex}} \quad \rightarrow_\beta \quad \underbrace{M[N/x]}_{\text{reduct}}$$

Lambda-Calculus as a Programming Language

As a *programming language*, the λ-calculus *computes* by repeatedly contracting redexes to their reducts:

$$
\begin{aligned}
& (\lambda x \,.\, y)\,(\underline{(\lambda z \,.\, z\,z)\,(\lambda w \,.\, w)}) \\
\rightarrow_\beta \quad & (\lambda x \,.\, y)\,(\underline{(\lambda w \,.\, w)\,(\lambda w \,.\, w)}) \\
\rightarrow_\beta \quad & \underline{(\lambda x \,.\, y)\,(\lambda w \,.\, w)} \\
\rightarrow_\beta \quad & y
\end{aligned}
$$

A term containing no β-redexes is called a **β-normal form.**

If M reduces to a normal form N in zero or more β-steps then M **β-evaluates to** N, written "$\mathrm{M} \downarrow_\beta \mathrm{N}$".

Beta-Reduction Relation

Formally, β is a *relation* of λ-terms that relates redexes to their reducts:

$$\beta : \Lambda \nrightarrow \Lambda \qquad \text{given by} \qquad \beta\,((\lambda x\,.\,M)\,N \dashrightarrow M[N/x])$$

It is customary to write this as:

$$\frac{}{(\lambda x\,.\,M)\,N \;\to_\beta\; M[N/x]}\;\beta$$

This relation is extended to make it compatible with application and abstraction:

$$\frac{M \;\to_\beta\; M'}{M\,N \;\to_\beta\; M'\,N}\;\beta\text{-app}_0 \qquad \frac{N \;\to_\beta\; N'}{M\,N \;\to_\beta\; M\,N'}\;\beta\text{-app}_1 \qquad \frac{M \;\to_\beta\; M'}{\lambda x\,.\,M \;\to_\beta\; \lambda x\,.\,M'}\;\beta\text{-abs}$$

The reflexive-transitive closure of β is the *preorder relation* β^* ("$\to_{\beta^*}$" or "$\twoheadrightarrow_\beta$").

The reflexive-symmetric-transitive closure is the *equivalence relation* $=_\beta$.

Evaluation Strategies

As a programming language, the λ-calculus is underspecified.

We could have done either

$$
\begin{aligned}
& (\lambda x \,.\, y)\, \underline{((\lambda z \,.\, z\, z)\, (\lambda w \,.\, w))} \\
\to_\beta \;& (\lambda x \,.\, y)\, \underline{((\lambda w \,.\, w)\, (\lambda w \,.\, w))} \\
\to_\beta \;& \underline{(\lambda x \,.\, y)\, (\lambda w \,.\, w)} \\
\to_\beta \;& y
\end{aligned}
$$

or

$$
\begin{aligned}
& \underline{(\lambda x \,.\, y)\, ((\lambda z \,.\, z\, z)\, (\lambda w \,.\, w))} \\
\to_\beta \;& y
\end{aligned}
$$

Try it yourself at `https://lambdacalc.io/`!

A choice of which redex(es) to reduce constitutes an **evaluation strategy**.

This brings up some questions:

- Do all strategies result in normal forms?
- Can different strategies give different normal forms?

Nontermination

Not all λ-terms have normal forms.

Consider the term

$$\Omega \;:=\; (\lambda x \,.\, x\, x)\,(\lambda x \,.\, x\, x)$$

This term has only one redex to reduce:

$$\begin{array}{ll} & \underline{(\lambda x \,.\, x\, x)\,(\lambda x \,.\, x\, x)} \\ \to_\beta & \underline{(\lambda x \,.\, x\, x)\,(\lambda x \,.\, x\, x)} \\ \to_\beta & \ldots \end{array}$$

Undecidability of Normalizability

Reducing a λ-term to a normal form is like running a Turing machine until it halts to see what it outputs.

We have seen that the *halting problem* for Turing machines is undecidable.

The situation for reducing λ-terms is just as bad.

Theorem (Turing)

The problem of deciding whether a λ-term has a normal form is undecidable.

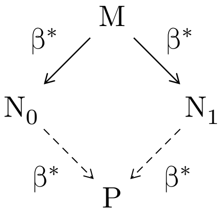
Confluence

Although in general we can't know whether a term has a normal form without trying to reduce it, the *evaluation strategy* we use to do so is not critical:

Theorem (Church & Rosser, Shroer)

The relation $\beta^* : \Lambda \nrightarrow \Lambda$ is **confluent** in the sense that for any term M,

$$\forall\, N_0, N_1 \,.\ \text{if } M \to_{\beta^*} N_0 \text{ and } M \to_{\beta^*} N_1 \text{ then } \exists\, P \,.\ N_0 \to_{\beta^*} P \text{ and } N_1 \to_{\beta^*} P.$$



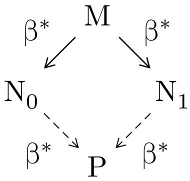
Uniqueness of Normal Forms

Corollary

If a normal form exists for a term then that normal form is unique.

Proof.

Suppose $\mathrm{M} \downarrow_\beta \mathrm{N}_0$ and $\mathrm{M} \downarrow_\beta \mathrm{N}_1$. Then by confluence:



the term P must be N_0 because N_0 is normal;
similarly, P must be N_1 because N_1 is normal.
So $\mathrm{N}_0 = \mathrm{N}_1$ by transitivity of equality. $\square$

Fixed Points

A **fixed point** of a function f is an argument x such that $f(x) = x$.

The function $f(x) := x^2$ has two fixed points, while $f(x) := x + 1$ has none.

In λ-calculus *every* term has a fixed point.

Moreover, there is a single term that can compute a fixed point of any term.

Curry's Fixed Point Combinator

The Y-**combinator** is the term $\mathrm{Y} := \lambda f \,.\, (\lambda x \,.\, f\,(x\,x))\,(\lambda x \,.\, f\,(x\,x))$.

Theorem

The Y-combinator computes a fixed point for any term M, in the sense that $\mathrm{M}\,(\mathrm{Y}\,\mathrm{M}) =_\beta \mathrm{Y}\,\mathrm{M}$.

Proof.

$$
\begin{array}{ll}
 & \mathrm{Y}\,\mathrm{M} \\
:= & \underline{(\lambda f \,.\, (\lambda x \,.\, f\,(x\,x))\,(\lambda x \,.\, f\,(x\,x)))\,\mathrm{M}} \\
\to_\beta & \underline{(\lambda x \,.\, \mathrm{M}\,(x\,x))\,(\lambda x \,.\, \mathrm{M}\,(x\,x))} \\
\to_\beta & \mathrm{M}\,(\underline{(\lambda x \,.\, \mathrm{M}\,(x\,x))\,(\lambda x \,.\, \mathrm{M}\,(x\,x))}) \\
\leftarrow_\beta & \mathrm{M}\,(\underline{(\lambda f \,.\, (\lambda x \,.\, f\,(x\,x))\,(\lambda x \,.\, f\,(x\,x)))\,\mathrm{M}}) \\
=: & \mathrm{M}\,(\mathrm{Y}\,\mathrm{M})
\end{array}
$$

□

Programming in λ-Calculus

We claimed that λ-calculus is a programming language.

So far we have just *pure functions*. Where are the

- booleans?
- numbers?
- tuples?
- lists?
- recursive functions?
- etc.

Amazingly, we can conjure them all out of pure functions.

Booleans

The following λ-terms are known as **Church booleans**:

$$
\begin{aligned}
\top &:= \lambda x\, y\,.\, x \\
\bot &:= \lambda x\, y\,.\, y \\
\neg &:= \lambda b\,.\, b \bot \top \\
\wedge &:= \lambda x\, y\,.\, x\, y \bot \\
\vee &:= \lambda x\, y\,.\, x \top y \\
\text{if} &:= \lambda b\, x\, y\,.\, b\, x\, y
\end{aligned}
$$

Experiment with evaluating boolean expressions at `https://lambdacalc.io/`.

Natural Numbers

The following λ-terms are known as **Church numerals**:

$$\begin{aligned} 0 &:= \lambda f\, x\, .\, x \\ \mathrm{S} &:= \lambda n\, f\, x\, .\, f\,(n\, f\, x) \\ + &:= \lambda m\, n\, .\, m\, \mathrm{S}\, n \\ \times &:= \lambda m\, n\, f\, .\, m\,(n\, f) \end{aligned}$$

Indeed, we can encode all the standard arithmetic functions in λ-calculus.

A tricky one is the **predecessor** $n - 1$, which stumped people for a long time.

Church's student Kleene cracked it while having his wisdom teeth extracted.

The trick is to encode ordered pairs $(m\, ,\, m + 1)$, increment them using S, and take the first projection when the second projection is n.

Recursive Functions

We can use *fixed points* to write **recursive functions.** Consider the factorial:

```
fact n  =  if (is0 n) 1 (× n (fact (pred n)))
fact  =  λ n . if (is0 n) 1 (× n (fact (pred n)))
fact  =  (λ f n . if (is0 n) 1 (× n (f (pred n)))) fact
```

Now `fact` is defined to be some function applied to `fact`. Call that function "`F`":

```
F := λ f n . if (is0 n) 1 (× n (f (pred n)))
```

Then `fact = F fact`. So `fact` is a fixed point of F.

```
fact  =  Y F
```

We can use this to evaluate `fact 2`.

"Let us calculate" – Leibniz

Recursive Functions in Python

This works in Python too, we just need to add one more layer of functions to interrupt Python's eager evaluation:

```
Y = lambda f : \
    (lambda x : f (lambda y : x (x) (y))) \
    (lambda x : f (lambda y : x (x) (y)))
```

Then we can write the non-recursive function of which factorial is a fixed point:

```
F = lambda f : lambda n : 1 if n == 0 else n * f (n - 1)
```

And define the recursive factorial function as its fixed point:

```
fact = Y (F)
```

to compute larger factorials quickly.

Lambda-Computable Functions

Theorem (Turing)

Every function $\mathbb{N} \to \mathbb{N}$ that is computable by Turing machine is computable in the λ-calculus.