"Sapienza" University of Rome
Faculty of Information Engineering,
Informatics and Statistics
Department of Computer Science

Models of Computation

Lecture notes integrated with the book TODO

Author
Alessio Bandiera

October 4, 2024

Contents

Information and Contacts

Personal notes and summaries collected as part of the *Models of Computation* course offered by the degree in Computer Science of the University of Rome "La Sapienza".

Further information and notes can be found at the following link:
https://github.com/aflaag-notes. Anyone can feel free to report inaccuracies, improvements or requests through the Issue system provided by GitHub itself or by contacting the author privately:

- Email: **alessio.bandiera02@gmail.com**
- LinkedIn: **Alessio Bandiera**

The notes are constantly being updated, so please check if the changes have already been made in the most recent version.

Suggested prerequisites:

- Linguaggi di Programmazione
- Tecniche di Programmazione Funzionale ed Imperativa

Licence:

1

TODO

1.1 TODO

1.1.1 TODO

In this first section, examples will be omitted from this notes, refer to the notes of the "Linguaggi di Programmazione" course for further details.

this notes are WIP, sections WILL change

Definition 1.1: Lambda calculus

Let Var be the set of all possible variables; thus, the **set Λ of all possible λ-terms** is defined by the following rules:

$$[var]\ \frac{x \in \text{Var}}{x \in \Lambda}$$

$$[appl]\ \frac{M \in \Lambda \quad N \in \Lambda}{MN \in \Lambda}$$

$$[abs]\ \frac{x \in \text{Var} \quad M \in \Lambda}{\lambda x.M \in \Lambda}$$

The terms of the form $\lambda x.M$ are called **λ-abstractions**, and MN is the function application of M to N. Note that function application *associates to the left*, therefore

$$MNL = (MN)L \neq M(NL)$$

Lambda calculus can be alternatively defined with the Backus Normal Form (BNF), as follows:

$$M, N ::= x \mid \lambda x.M \mid MN$$

Although *all functions in lambda calculus are unary*, the following definition can expand this concept.

Definition 1.2: Currying

Currying (named after Haskell Curry) is defined as follows:

$$\lambda x_1.(\dots(\lambda x_n.y)) \equiv \lambda x_1 \dots x_n.y$$

Definition 1.3: Boundness

A variable is said to be **bound** if it is declared in a λ-abstraction, otherwise it is said to be **free**.

A term that has no free variables is said to be **closed** or **combinator**.

Example 1.1 (Boundness)**.** Consider the following term:

$$\lambda x.xy$$

In this example, x is *bound*, and y is *free*.

Definition 1.4: Notable combinators

The following are some of the **notable combinators**:

$$\begin{aligned}
\mathrm{I} &\equiv \lambda x.x \\
\mathrm{K} \equiv T &\equiv \lambda xy.x \\
\mathrm{O} \equiv F &\equiv \lambda xy.y \\
\mathrm{S} &\equiv \lambda xyz.xz(yz) \\
\mathrm{B} &\equiv \lambda fgx.f(gx) \\
\mathrm{C} &\equiv \lambda abc.acb \\
\mathrm{W} &\equiv \lambda xy.xyy
\end{aligned}$$

Definition 1.5: Free variables

Given a λ-term, the function

$$\mathrm{free} : \Lambda \to \mathcal{P}(\mathrm{Var})$$

returns the **set of free variables in** M, and it is defined recursively as follows:

$$\begin{cases}
\mathrm{free}(x) := \{x\} \\
\mathrm{free}(MN) := \mathrm{free}(M) \cup \mathrm{free}(N) \\
\mathrm{free}(\lambda x.M) := \mathrm{free}(M) - \{x\}
\end{cases}$$

Definition 1.6: Substitution

The **substitution** operation is recursively defined by the following rules:

$$x[N/x] = N$$

$$y[N/x] = y$$

$$(PQ)[N/x] = P[N/x]\ Q[N/x]$$

$$(\lambda t.P)[N/x] = \lambda t.(P[N/x])$$

where $M[N/x]$ means that *each instance of x in M is replaced with N*. Note that **only free variables may be substituted**.

Definition 1.7: Inference rules

The following are the **inference rules** for the lambda caluclus:

$$(\alpha)\ \ \lambda x.M \equiv (\lambda y.M)[y/x]$$

$$(\beta)\ \ (\lambda x.M)N \ \xrightarrow{\beta}\ M[N/x]$$

$$(\mu)\ \ \frac{M \ \xrightarrow{\beta}\ M'}{NM \ \xrightarrow{\beta}\ NM'}$$

$$(\nu)\ \ \frac{M \ \xrightarrow{\beta}\ M'}{MN \ \xrightarrow{\beta}\ M'N}$$

$$(\xi)\ \ \frac{M \ \xrightarrow{\beta}\ M'}{\lambda x.M \ \xrightarrow{\beta}\ \lambda x.M'}$$

Note that the β-rule is effectively *one step of the computation* described by the λ-term.

Definition 1.8: Normal form

If a term can be β-reduced, it is called **β-redex**, or simply **redex** (*reducible expression*), and the reduced term is called **β-reduct**, or simply **reduct**.

If a term has no redexes, it is said to be in **normal form**.

Observation 1.1: Variable capture

Consider the following λ-term:

$$(\lambda xt.tx)(\lambda t.y) \xrightarrow{\beta} \lambda t.t(\lambda t.y)$$

Note that the two ts are *different*. In fact, underlining the λ-abstractions to which they are bounded to can help clarifying their distinction:

$$(\lambda x\underline{t}.\underline{t}x)(\lambda t.y) \xrightarrow{\beta} \lambda \underline{t}.\underline{t}(\lambda t.y)$$

Now, consider the following λ-term, similar to the previous one:

$$(\lambda xy.yx)(\lambda t.y) \xrightarrow{\beta} \lambda y.y(\lambda t.y)$$

This β-reduction created a problem, because now the two ys *are the same*, even though they were not originally. In fact, the previous term can be relabeled as follows:

$$(\lambda x\underline{y}.\underline{y}x)(\lambda t.y) \xrightarrow{\beta} \lambda \underline{y}.\underline{y}(\lambda t.\underline{y})$$

This happened because

$$\text{free}(\lambda t.y) = \{y\} - \{t\} = \{y\}$$

therefore y was **captured** by the y that was already present in the leftmost λ-abstraction. This phenomena is called **variable capturing**, and constitutes a problem when reducing β-redexes. In particular, to reduce this second λ-abstraction, it is necessary to apply a substitution, by using the α rule (refer to Definition 1.7):

$$\lambda xy.yx = \lambda x(\lambda y.yx) = \lambda x.((\lambda y.yx)[u/y]) = \lambda x.(\lambda u.ux) = \lambda xu.ux$$

which means that the β-reduction can now be performed without any issue:

$$(\lambda xu.ux)(\lambda t.y) \xrightarrow{\beta} \lambda u.u(\lambda t.y)$$

where y is still free. Note that it would not have been *safe* to rename the other (free) y, because in general *renaming free variables can create capturing problems as well*. For example, y could have not been substituted with t, as it would otherwise be captured by the t in the λ-term $\lambda t.y$, as follows:

$$(\lambda t.y)[t/y] = \lambda t.t$$

Fortunately, variable capturing can be solved by employing the following *variable naming convention*.

Definition 1.9: Variable naming convention

To avoid variable capturing problems, it is sufficient to follow this **variable naming convention**: *bound and free variables must have different names between them.*

From now on, it will be assumed that any β-reduction is performed by renaming opportunely the **bound** variables, such that in each step of the computation the naming convention is followed.

Definition 1.10: Tuples

A **tuple** of the form
$$(M_1, \dots, M_k)$$
can be represented in λ-calculus as follows:
$$\lambda x.xM_1 \dots M_k$$

To access the elements of a tuple, *projectors* are used, which are defined below

Definition 1.11: Projector

A **projector** has the following form
$$\lambda x.x\pi_j^k$$
where
$$\pi_j^k \equiv \lambda x_1 \dots x_k.x_j$$

Example 1.2 (Projectors)**.** Given a tuple $\lambda x.xM_1 \dots M_k$, its j-th element can be accessed as follows:
$$\lambda x.x\pi_j^k(\lambda x.xM_1 \dots M_k) \xrightarrow{\beta} (\lambda x.xM_1 \dots M_k)\pi_j^k \xrightarrow{\beta} \pi_j^k M_1 \dots M_k \xrightarrow{\beta} M_j$$

Definition 1.12: Fixed point

Given a function $f : X \to Y$, an element $x \in X$ is said to be a **fixed point of** f if and only if $f(x) = x$.

Example 1.3 (Fixed points)**.** Given a function $f(x) = x^2 - 3x + 4$, $x = 2$ is a *fixed point* of f, because
$$f(x) = 2^2 - 3 \cdot 2 + 4 = 4 - 6 + 4 = 2 = x$$
and thus $f(x) = x$.

Example 1.4 (Functions are fixed points)**.** Consider the following function
$$F(g) :\equiv h(x) = \begin{cases} 1 & x = 0 \\ x \cdot g(x-1) & x > 0 \end{cases}$$

that takes a function as input, an returns a function h; for instance, plugging in the following function

$$\text{succ} : x \to x + 1$$

we get that F returns the following function

$$F(\text{succ}) \equiv h(x) = \begin{cases} 1 & x = 0 \\ x \cdot \text{succ}(x-1) = x \cdot x = x^2 & x > 0 \end{cases}$$

which is the function that returns 1 if $x = 0$, and x^2 otherwise.

It's easy to check that the *fixed point* of F is the following function:

$$\text{fact}(x) := \begin{cases} 1 & x = 0 \\ x \cdot \text{fact}(x-1) & x > 0 \end{cases}$$

which computes the factorial of x, because

$$F(\text{fact}) \equiv h(x) = \begin{cases} 1 & x = 0 \\ x \cdot \text{fact}(x-1) & x > 0 \end{cases} \equiv \text{fact}$$

Definition 1.13: Kleene's combinator

The **fixed point operator**, Y **combinator** or **Kleene's combinator** (named after Stephen Kleene) is defined as follows:

$$\text{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

The Y combinator can be alternatively defined as follows:

$$\text{Y} \equiv (\lambda xy.y(xxy))(\lambda xy.y(xxy))$$

Proposition 1.1: Fixed point operator

Given a function, Kleene's combinator returns its fixed point.

Proof. If the Kleene's combinator can return the fixed point of a given function h, it means that $\text{Y}h$ is h's fixed point. Therefore, the statement that has to be proved is that

$$h(\text{Y}h) \equiv \text{Y}h$$

This can be proved for both formulations of the Y combinator, as follows:

$$\begin{aligned} \text{Y}h &\xrightarrow{\beta} (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))h \\ &\xrightarrow{\beta} (\lambda x.h(xx))(\lambda x.h(xx)) \\ &\xrightarrow{\beta} h(\lambda x.h(xx)\lambda x.h(xx)) \\ &\xrightarrow{\beta} h(\text{Y}h) \end{aligned}$$

and for the alternative formulation

$$
\begin{aligned}
\mathrm{Y}h &\xrightarrow{\beta} ((\lambda xy.y(xxy))(\lambda xy'.y'(xxy')))h \\
&\xrightarrow{\beta} (\lambda y.y((\lambda xy'.y'(xxy'))(\lambda xy''.y''(xxy''))y))h \\
&\xrightarrow{\beta} h((\lambda xy'.y'(xxy'))(\lambda xy''.y''(xxy''))h) \\
&\xrightarrow{\beta} h(\mathrm{Y}h)
\end{aligned}
$$

□

Note that the Y combinator can be used to perform *recursion* inside λ-calculus, because of the following property:

$$
\begin{gathered}
h(\mathrm{Y}h) = \mathrm{Y}h \\
h(h(\mathrm{Y}h)) = h(\mathrm{Y}h) = \mathrm{Y}h \\
\vdots \\
h(\ldots(h(\mathrm{Y}h))) = \mathrm{Y}h
\end{gathered}
$$

should i go deeper?

1.2 Exercises

Problem 1.1: Solve for X

Find X such that

$$Xx = \lambda t.t(Xx)$$

Solution. The term is

$$X \equiv (\lambda fbt.t(fb))X$$

because

$$
\begin{aligned}
Xx &\xrightarrow{\beta} (\lambda fbt.t(fb))Xx \\
&\xrightarrow{\beta} (\lambda bt.t(Xb))x \\
&\xrightarrow{\beta} \lambda t.t(Xx)
\end{aligned}
$$

Problem 1.2: Solve for H

Find H such that

$$H(\lambda x_1x_2x_3.P) = \lambda x_3x_2x_1.P$$

Solution. The term is

$$H \equiv \lambda fx_3x_2x_1.fx_1x_2x_3$$

because

$$\begin{aligned} H(\lambda x_1 x_2 x_3.P) &\xrightarrow{\beta} (\lambda f x_3 x_2 x_1.f x_1 x_2 x_3)(\lambda x_1 x_2 x_3.P) \\ &\xrightarrow{\beta} \lambda x_3 x_2 x_1.(\lambda x_1 x_2 x_3.P) x_1 x_2 x_2 \\ &\xrightarrow{\beta} \lambda x_3 x_2 x_1.P \end{aligned}$$