"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

November 2, 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 λ-caluclus

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

Definition 1.1: λ-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:

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

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

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

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:

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

Additionally, the following notation

$$\lambda \overrightarrow{x}.f\ \overrightarrow{x}$$

will denote **vectors** in place of

$$\lambda x_1 \ldots x_n.f\ x_1 \ldots x_n$$

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**:

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

In particular

- I stands for “Identity”, in fact it is the identity function
- K stands for “Konstant”, meaning *constant* in german, and in fact K yields a constant function

- S stands for "Satz", which translates to *substitution* in german, and plays a central role in Section 1.1.5
- B stands for "Belegung", meaningn *binding* in german, in fact B is the composition operator
- C stands for "Commutator", which is precisely what the C combinator performs
- W is the duplicating combinator, and it is called W because the letter W resambles the shape of two V's placed side by side

Definition 1.5: Free variables

Given a λ-term, the function
$$\text{free} : \Lambda \to \mathcal{P}(\text{Var})$$
returns the **set of free variables in** M, and it is defined recursively as follows:

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

1.1.1 β-reduction

Definition 1.6: Substitution

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

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

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

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

$$(\lambda y.P)[N/x] = \lambda y.(P[N/x]) \text{ if } y \neq x$$

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

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

Lemma 1.1: Substitution lemma

Let $M, N, L \in \Lambda$; if $x \neq y$ and $x \notin \text{free}(L)$, then

$$M[M/x][L/y] \equiv M[L/y][N[L/y]/x]$$

Proof. By induction on the structure of M, the details are omitted. □

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.

If $M \equiv N$ is provable in the λ-caluclus, it will be written as

$$\lambda \vdash M \equiv N$$

Additionally, if a term N can be derived from M through β-reductions

$$M \ \xrightarrow{\beta}\ \dots \ \xrightarrow{\beta}\ N$$

it will be written as $M \rightsquigarrow N$.

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.

1.1.2 Data structures

Definition 1.10: Tuples

A **tuple** of the form

$$(M_1, \dots, M_k)$$

can be represented in λ-calculus as follows:

$$\lambda x.xM_1 \dots M_k$$

In λ-caluclus, tuples will be represented as follows

$$[M_1, \dots, M_k]$$

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: Booleans

Booleans can be defined in λ-calculus as follows:

$$\mathrm{T} \equiv \lambda xy.x$$
$$\mathrm{F} \equiv \lambda xy.y$$

Definition 1.13: Conditionals

Conditionals can be defined in λ-calculus as follows:

$$\text{ite} = \lambda xyz.xyz$$

Note that ite stands for *"if-then-else"*, and in fact, the term behaves exactly like a condition when used in conjunction with the λ-booleans.

Observation 1.2: Conditionals

The term ite correctly behaves as a *conditional* when used with T and F. In fact, when used in an term such as

$$\text{ite C A B}$$

if C is a λ-boolean, the term with be β-reduced to A if C ≡ T, and it will be evaluated to B if C ≡ F. Indeed

$$\begin{aligned} \text{ite T A B} &\equiv (\lambda xyz.xyz) \text{ T A B} \\ &\xrightarrow{\beta} \text{T A B} \\ &\equiv (\lambda xy.x) \text{ A B} \\ &\xrightarrow{\beta} \text{A} \end{aligned}$$

and

$$\begin{aligned} \text{ite F A B} &\equiv (\lambda xyz.xyz) \text{ F A B} \\ &\xrightarrow{\beta} \text{F A B} \\ &\equiv (\lambda xy.y) \text{ A B} \\ &\xrightarrow{\beta} \text{B} \end{aligned}$$

Definition 1.14: Church numerals

The **Church numerals** are defined by a mapping between natural numbers $\mathbb{N}$ and λ-abstractions:

$$\varphi : \mathbb{N} \to \Lambda : n \mapsto \lambda xy.\, \underbrace{x(\dots(x}_{n\text{ times}}\, y))$$

The Church numeral corresponding to $n \in \mathbb{N}$ will be represented as $\underline{n} \in \Lambda$.

Example 1.3 (Church numerals)**.** For example, the number $\underline{0}$ is represented as

$$\varphi(0) = \underline{0} = \lambda xy.y \equiv \text{F} \equiv \text{O}$$

number $\underline{1}$ as

$$\varphi(1) = \underline{1} = \lambda xy.xy \equiv \text{T} \equiv \text{K}$$

and number $\underline{2}$ as

$$\varphi(2) = \underline{2} = \lambda xy.x(xy)$$

and so on.

Observation 1.3: Church numerals are iterators

Note that Church numerals are **iterators**, in the sense that $\underline{n}$ replicates any input function f for n times

$$\begin{aligned} \underline{n}\ f\ \chi &\equiv (\lambda xy.\, \underbrace{x(\dots(x\, y)}_{n \text{ times}}))\ f\ \chi \\ &\xrightarrow{\beta}\ \underbrace{f(\dots(f\, \chi)}_{n \text{ times}}) \end{aligned}$$

The following are some important λ-functions for Church numerals:

- **successor function**:

  $$\underline{s} \equiv \lambda xyz.xy(yz)$$

  which given a Church numeral $\underline{n}$, it returns $\underline{n+1}$, which is easy to show

  $$\begin{aligned} \underline{s}\ \underline{n} &\equiv (\lambda abc.ab(bc))\ (\lambda xy.\, \underbrace{x(\dots(x\, y)}_{n \text{ times}})) \\ &\xrightarrow{\beta}\ \lambda bc.(\lambda xy.\, \underbrace{x(\dots(x\, y)}_{n \text{ times}})\ b\ (bc)) \\ &\xrightarrow{\beta}\ \lambda bc.(\lambda y.\, \underbrace{b(\dots(b\, y)}_{n \text{ times}})\ (bc)) \\ &\xrightarrow{\beta}\ \lambda bc.\ \underbrace{b(\dots(b\ c)}_{n+1 \text{ times}}) \end{aligned}$$

- **is-zero function**:

  $$\underline{z} \equiv \lambda f.f(\lambda t.\mathrm{F})\mathrm{T}$$

  which given a Church numeral $\underline{n}$, it returns T if and only if $\underline{n}$ is $\underline{0}$, and it can be proven at follows

  $$\begin{aligned} \underline{z}\ \underline{0} &\equiv (\lambda f.f(\lambda t.\mathrm{F})\mathrm{T})\ (\lambda xy.y) \\ &\xrightarrow{\beta}\ (\lambda xy.y)(\lambda t.\mathrm{F})\mathrm{T} \\ &\xrightarrow{\beta}\ \mathrm{T} \end{aligned}$$

and

$$\begin{aligned}
\underline{z}\ \underline{n} &\equiv (\lambda f.f(\lambda t.\mathrm{F})\mathrm{T})\ (\lambda xy.\underbrace{x(\dots(x}_{n\text{ times}}y)))\\
&\xrightarrow{\beta} (\lambda xy.\underbrace{x(\dots(x}_{n\text{ times}}y)))(\lambda t.\mathrm{F})\mathrm{T}\\
&\xrightarrow{\beta} \underbrace{(\lambda t.\mathrm{F})(\dots((\lambda t.\mathrm{F})}_{n\text{ times}}\mathrm{T}))\\
&\xrightarrow{\beta} \mathrm{F}
\end{aligned}$$

- **addition function**:

  $$\mathrm{add} \equiv \lambda ab.a\ \underline{s}\ b$$

  a proof of this function is omitted, but it can be intuitively explained by using Observation 1.3, which suggests that if $\underline{a}$ and $\underline{b}$ are two Church numerals, then $\underline{a}\ \underline{s}\ \underline{b}$ is the repeated application of $\underline{s}$ exactly a times to $\underline{b}$

- **predecessor function**:

  $$\mathrm{P} \equiv \lambda x.x(\lambda abz.zb(\underline{s}\ b))$$

  which is a term such that

  $$\begin{aligned}
  \mathrm{P}\ [\underline{m},\underline{n}] &\equiv (\lambda x.x(\lambda abz.zb(\underline{s}\ b)))\ (\lambda u.u\ \underline{m}\ \underline{n})\\
  &\xrightarrow{\beta} (\lambda u.u\ \underline{m}\ \underline{n})(\lambda abz.zb(\underline{s}\ b))\\
  &\xrightarrow{\beta} (\lambda abz.zb(\underline{s}\ b))\ \underline{m}\ \underline{n}\\
  &\xrightarrow{\beta} \lambda z.z\ \underline{n}\ (\underline{s}\ \underline{n})\\
  &\equiv [\underline{n},\underline{n+1}]
  \end{aligned}$$

  therefore

  $$\begin{aligned}
  \underline{n}\ \mathrm{P}\ [\underline{0},\underline{0}] &\xrightarrow{\beta} \underbrace{\mathrm{P}(\dots(\mathrm{P}}_{\text{n times}}\ [\underline{0},\underline{0}]))\\
  &\xrightarrow{\beta} \underbrace{\mathrm{P}(\dots(\mathrm{P}}_{\text{n}-1\text{ times}}\ [\underline{0},\underline{1}]))\\
  &\xrightarrow{\beta} \underbrace{\mathrm{P}(\dots(\mathrm{P}}_{\text{n}-2\text{ times}}\ [\underline{1},\underline{2}]))\\
  &\xrightarrow{\beta} \dots\\
  &\xrightarrow{\beta} [\underline{n-1},\underline{n}]
  \end{aligned}$$

  which means that the following term

  $$\underline{p} \equiv \lambda x.\mathrm{T}(x\ \mathrm{P}\ [\underline{0},\underline{0}])$$

  is such that

  $$\underline{p}\ \underline{n} \leadsto \underline{n-1}$$

Church numerals are not the only way to define a system of numerals in λ-calculus. In fact, Hank Barendregt proposed the following system in 1976.

Definition 1.15: Barendregt numerals

The Barendregt numerals are defined inductively as follows:

$$\ulcorner 0 \urcorner \equiv \mathrm{I}$$
$$\ulcorner n+1 \urcorner \equiv [\mathrm{F}, \ulcorner n \urcorner]$$

Therefore, we have that

$$\ulcorner n \urcorner \equiv \underbrace{[\mathrm{F}, \dots, [\mathrm{F}}_{\text{n times}}, \mathrm{I}]]$$

From this definition, the following λ-function for Barendregt numerals can be defined:

- **successor function**:
$$\mathbf{s} \equiv \lambda x.x[\mathrm{F}, x]$$
- **predecessor function**:
$$\mathbf{f} \equiv \lambda x.x\ \mathrm{F}$$
which consumes the outermost tuple
- **is-zero function**:
$$\mathbf{z} \equiv \lambda x.x\ \mathrm{T}$$
because it returns F, the first element of any tuple, for any Barendregt numeral, except when the input is not a tuple but I itself, which is precisely $\ulcorner 0 \urcorner$

1.1.3 Recursion

Definition 1.16: 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 1.4 (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 1.5 (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 1.17: Kleene's combinator

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

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

The Y combinator can be alternatively defined as follows:

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

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 $\mathrm{Y}h$ is h's fixed point. Therefore, the statement that has to be proved is that

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

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

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

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:

$$\begin{aligned} h(\mathrm{Y}h) &= \mathrm{Y}h \\ h(h(\mathrm{Y}h)) = h(\mathrm{Y}h) &= \mathrm{Y}h \\ &\vdots \\ h(\dots(h(\mathrm{Y}h))) &= \mathrm{Y}h \end{aligned}$$

1.1.4 Recursive functions

Definition 1.18: Numeric function

A **numeric function** is a map $f : \mathbb{N}^p \to \mathbb{N}$ for some p.

Definition 1.19: λ-definable function

A function is said to be **λ-definable** if there exists a closed term F such that

$$F\ \underline{n_1} \dots \underline{n_p} \equiv \underline{f(n_1, \dots, n_p)}$$

If that is the case, f is said to be **λ-defined** by F.

Definition 1.20: Initial functions

The following are the so called **initial functions**:

$$\begin{aligned} U_r^i(x_1, \dots, x_r) &= x_i, \quad 1 \le i \le r \\ \mathrm{succ}(n) &= n + 1 \\ \mathrm{zero}(n) &= 0 \end{aligned}$$

In particular, the first equations are called **projection functions**, the second is the **successor function** and the last is called **constant function**.

Definition 1.21: Composition

Given an m-ary function $h(x_1, \dots, x_m)$ and k m-ary functions

$$g_1(x_1, \dots, x_k), \dots, g_m(x_1, \dots, x_k)$$

, the **composition operator** is defined as follows:

$$f := h \circ (g_1, \dots, g_m)$$

where

$$f(\overrightarrow{x}) = h(g_1(\overrightarrow{x}), \dots, g_m(\overrightarrow{x}))$$

Lemma 1.2: Composition

The λ-definable functions are closed under *composition*.

Proof. Let $h, g_1, \dots, g_m$ be defined by the λ-terms $H, G_1, \dots, G_m$ respectively. Then

$$F \equiv \lambda \overrightarrow{x}.H(G_1 \overrightarrow{x}) \dots (G_m \overrightarrow{x})$$

λ-defines $h \circ (g_1, \dots, g_m)$. □

Definition 1.22: Primitive recursion

Given a k-ary function $g(x_1, \dots, x_k)$ and a $(k+2)$-ary function $h(y, z, x_1, \dots, x_k)$, the **primitive recursion operator** is a $(k+1)$-ary function ρ is defined as follows:

$$f := \rho(g, h)$$

where

$$f(0, \overrightarrow{n}) = g(\overrightarrow{x})$$
$$f(\mathrm{succ}(n), \overrightarrow{x}) = h(y, f(y, \overrightarrow{x}), \overrightarrow{x})$$

Lemma 1.3: Primitive recursion

The λ-definable functions are closed under *primitive recursion*.

Proof. Let f be a function such that

$$f(0) = g$$
$$f(k+1) = h(f(k), k)$$

which is a weaker version of the *primitive recursion operator*, but the proof for general $\overrightarrow{n}$ is similar (details are omitted).

Consider the following term

$$T \equiv \lambda p.[\underline{s}(p\ \mathrm{T}), H(p\ \mathrm{F})(p\ \mathrm{T})]$$

where H λ-defines h, and note how it computes over the following input

$$\begin{aligned} T\ [\underline{k}, \underline{f(x)}] &\rightsquigarrow [\underline{s}\ \underline{k}, H\ \underline{f(x)}\ \underline{k}] \\ &\xrightarrow{\beta} [\underline{k+1}, H\ \underline{f(x)}\ \underline{k}] \\ &\equiv [\underline{k+1}, \underline{f(k+1)}] \end{aligned}$$

(the last step follows from f's definition) therefore, T describes the following function

$$t : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N} : (k, f(k)) \mapsto (k+1, f(k+1))$$

Hence, by induction, it follows that

$$[\underline{k}, \underline{f(k+1)}] \equiv T^k\ [\underline{0}, \underline{f(0)}]$$

and because Church numerals are iterators (as shown in Observation 1.3), it follows that

$$\underline{f(k)} \equiv \underline{k}\ T\ [\underline{0}, \underline{f(0)}]\ \mathrm{F}$$

Finally, f can be λ-defined by the following term

$$F \equiv \lambda k.k\ T\ [\underline{0}, \underline{G}]\ \mathrm{F}$$

where G λ-defines g. □

Definition 1.23: Minimalization

Given a $(k+1)$-ary function $f(y, x_1, \ldots, x_k)$, the **minimization operator** is a k-ary function $\mu(f)$ defined as follows:

$$\mu(f)(\vec{x}) = z \iff \forall i \in [0, z-1] \quad f(i, \vec{x}) > 0 \land f(z, \vec{x}) = 0$$

In other words, that *minimization operator* seeks the smallest argument that causes the function to return zero. If there is no such argument, or if an argument is encountered for which f is not defined, then the search never terminates, and $\mu(f)$ is not defined for $\vec{x}$.

Lemma 1.4: Minimalization

The λ-definable functions are closed under minimalization

Proof. TODO □

Definition 1.24: Recursive functions

The class $\mathcal{R}$ is the smallest class of *numeric functions* that contains *all initial functions*, and is closed under *composition*, *primitive recursion* and *minimization*.

Theorem 1.1: Recursive functions

All recursive functions are λ-definable, therefore

$$\mathcal{R} \subsetneq \Lambda$$

Proof. The following terms λ-define *initial functions*:

$$\begin{aligned} U^i_p &\equiv \pi^k_j \\ \text{succ} &\equiv \underline{s} \\ \text{zero} &\equiv \underline{0} \end{aligned}$$

Thus, the theorem follows directly from Lemma 1.2, Lemma 1.3 and Lemma 1.4. □

1.1.5 Combinatory Logic

Definition 1.25: Combinatory Logic

Combinatory Logic is a simplified model of computation, related to λ-calculus, defined as follows:

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

$$[appl]\ \frac{U \in \Lambda \quad V \in \Lambda}{UV \in \text{CL}}$$

$$[const]\ \frac{x \in \text{Const}}{x \in \text{CL}}$$

where the set Const is defined by the following **constants**:

$$\text{Const} := \{\text{S}, \text{K}, \text{I}, \text{B}, \text{C}\}$$

Association rules are the same as the ones for λ-calculus. Note that, in combinatory logic, there are **no abstractions**, therefore *all variables are free*. To compute with CL, **reductions** are defined as follows:

$$\begin{aligned} \text{S}xys &\rhd xy(yz) \\ \text{K}xy &\rhd x \\ \text{I}x &\rhd x \\ \text{B}xyz &\rhd x(yz) \\ \text{C}xyz &\rhd xzy \end{aligned}$$

Theorem 1.2: Equivalence of λ-calculus and CL

Every λ-term can be written as a CL term, and viceversa.

Proof. Given a CL term $U \in \text{CL}$:

- if $U \in \text{CL}$ because $U \in \text{Var}$, then $U \in \Lambda$ by definition of Λ
- if $U \in \text{CL}$ because $U \equiv (MN)$ where $M, N \in \text{CL}$, then $(MN) \in \Lambda$ because applications are defined in λ-caluclus as well, and $M, N \in \Lambda$ inductively
- finally, if $U \in \text{CL}$ because $U \in \text{Const}$, then simply convert the CL constant into a λ-combinator (refer to Definition 1.4)

This proves that $\text{CL} \subseteq \Lambda$. To show the other inclusion, a similar reasoning can be applied, except for λ-abstractions, which do not exist in CL. Nevertheless, the following recursive algorithm returns a CL term equivalent to any given λ-abstraction. Let the *swap* operation be defined as follows:

$$\text{swap} : \Lambda_{abstr} \to \text{CL} : \lambda x.P \mapsto [x]P$$

then, the conversion algorithm defines additional rules to the *swap* operation as follows:

$$\begin{aligned}
[x]Ux &= U \\
[x]x &= \text{I} \\
[x]U &= \text{K}U \\
[x]UW &= \text{B}U([x]W) \\
[x]VU &= \text{C}([x]V)U \\
[x]VW &= \text{S}([x]V)([x]W)
\end{aligned}$$

where $U, V, W \in \text{CL}$ are three CL terms such that $x \notin \text{free}(U)$ and $x \in \text{free}(V), \text{free}(W)$. □

An intuition of the correctness of the algorithm can be provided as follows (bounded variables are renamed differently from Definition 1.4, to match the definitions in the algorithm):

- the first rule is a special case
- $\text{swap}(\text{I}) = \text{swap}(\lambda x.x) = [x]x$
- $\text{swap}(\text{K}U) = \text{swap}(\lambda x.U) = [x]U$
- $\text{swap}(\text{B}U(\lambda a.W)) = \text{swap}(\lambda x.U(\lambda a.W)x) = [x]UW$
- $\text{swap}(\text{C}(\lambda a.V)U) = \text{swap}(\lambda x.(\lambda a.V)xU) = [x]VU$
- $\text{swap}(\text{S}(\lambda a.V)(\lambda b.W)) = \text{swap}(\lambda x.(\lambda a.V)x(\lambda(b.W)x)) = [x]VW$

Example 1.6 (Conversion Λ to CL)**.** Consider the following λ-term

$$\lambda xy.ytx$$

it can be converted into a CL term as follows

$$\begin{aligned}
\mathrm{swap}(\lambda xy.ytx) &= \mathrm{swap}([x](\lambda y.ytx)) \\
&= [x]([y](yt)x) \\
&= [x](\mathrm{C}([y]yt)x) \\
&= [x](\mathrm{C}(\mathrm{C}([y]y)t)x) \\
&= [x]\mathrm{C}(\mathrm{CI}t)x \\
&= \mathrm{C}(\mathrm{CI}t)
\end{aligned}$$

and in fact

$$\begin{aligned}
\mathrm{C}(\mathrm{CI}t)xy &\rhd \mathrm{CI}tyx \\
&\rhd \mathrm{I}yt \\
&\rhd ytx
\end{aligned}$$

Because CL and Λ are equivalent, Church numerals can be defined in CL as well:

- $\mathrm{swap}(\underline{0}) = \mathrm{swap}(\lambda xy.y) = [x][y]y = [x]\mathrm{I} = \mathrm{KI}$ and in fact $\mathrm{KI}xy \rhd \mathrm{I}y \rhd y$
- $\mathrm{swap}(\underline{1}) = \mathrm{swap}(\lambda xy.xy) = [x][y]xy = [x]x = \mathrm{I}$ and in fact $\mathrm{I}xy \rhd xy$
- $\mathrm{swap}(\underline{2}) = \mathrm{swap}(\lambda xy.x(xy)) = [x][y]x(xy) = [x]\mathrm{B}x([y]xy) = [x]\mathrm{B}xx = \mathrm{S}([x]\mathrm{B}x)([x]x) = \mathrm{SBI}$ and in fact $\mathrm{SBI}xy \rhd \mathrm{B}x(\mathrm{I}x)y \rhd \mathrm{B}xxy \rhd x(xy)$
- in general, we have that
$$\mathrm{swap}(\underline{n}) = \underbrace{\mathrm{SB}(\dots(\mathrm{SB}}_{\text{n}-1\ \text{times}}\mathrm{I}))$$

1.2 Exercises

Problem 1.1: Solve for X

Find X such that
$$Xx = \lambda t.t(Xx)$$

Solution. The term is

$$X \equiv (\lambda fbt.t(fb))X \implies X \equiv \mathrm{Y}(\lambda fbt.t(fb))$$

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
$$H \equiv \lambda f x_3x_2x_1.fx_1x_2x_3$$
because
$$\begin{aligned} H(\lambda x_1x_2x_3.P) &\xrightarrow{\beta} (\lambda f x_3x_2x_1.fx_1x_2x_3)(\lambda x_1x_2x_3.P) \\ &\xrightarrow{\beta} \lambda x_3x_2x_1.(\lambda x_1x_2x_3.P)x_1x_2x_2 \\ &\xrightarrow{\beta} \lambda x_3x_2x_1.P \end{aligned}$$

Problem 1.3: Solve for X

Find X such that
$$Xxyz = Xz(uv)$$

Solution. The term is
$$X \equiv (\lambda tabc.tc(uv))X \implies X \equiv \mathrm{Y}(\lambda tabc.tc(uv))$$
because
$$\begin{aligned} (\lambda tabc.tc(uv))Xxyz &\xrightarrow{\beta} (\lambda abc.Xc(uv))xyz \\ &\xrightarrow{\beta} Xz(uv) \end{aligned}$$

Problem 1.4: Solve for Δ

Find Δ such that
$$\begin{cases} \Delta\mathrm{S} = y_1 \\ \Delta\mathrm{K} = y_2 \\ \Delta\mathrm{I} = y_3 \end{cases}$$

Solution. Assume that
$$\Delta \equiv \lambda x.xP_1P_2P_3$$
for some λ-terms P_1, P_2 and P_3; then
$$\begin{cases} \Delta\mathrm{S} \xrightarrow{\beta} \mathrm{S}\ P_1P_2P_3 \xrightarrow{\beta} P_1P_3(P_2P_3) \\ \Delta\mathrm{K} \xrightarrow{\beta} \mathrm{K}\ P_1P_2P_3 \xrightarrow{\beta} P_1P_3 \\ \Delta\mathrm{I} \xrightarrow{\beta} \mathrm{I}\ P_1P_2P_3 \xrightarrow{\beta} P_1P_2P_3 \end{cases}$$
However, this cannot be a correct assumption, because if
$$\Delta\mathrm{K} \xrightarrow{\beta} P_1P_3 = y_2$$

then

$$\Delta \mathrm{S} \xrightarrow{\beta} P_1P_3(P_2P_3) = y_2(P_2P_3) \neq y_1$$

which means that ΔS cannot be evaulated to y_1. This issue can be solved by assuming that

$$\Delta \equiv \lambda x.xP_1P_2P_3P_4$$

for some other term λ-term P_4, in fact

$$\begin{cases} \Delta \mathrm{S} \xrightarrow{\beta} \mathrm{S}\ P_1P_2P_3P_4 \xrightarrow{\beta} P_1P_3(P_2P_3)P_4 \\ \Delta \mathrm{K} \xrightarrow{\beta} \mathrm{K}\ P_1P_2P_3P_4 \xrightarrow{\beta} P_1P_3P_4 \\ \Delta \mathrm{I} \xrightarrow{\beta} \mathrm{I}\ P_1P_2P_3P_4 \xrightarrow{\beta} P_1P_2P_3P_4 \end{cases}$$

and if $P_1 = \lambda xy.y$ then

$$\Delta \mathrm{K} \xrightarrow{\beta} P_1P_3P_4 \xrightarrow{\beta} (\lambda xy.y)P_3P_4 \xrightarrow{\beta} P_4$$

whch means that P_4 must be y_2. Moreover

$$\Delta \mathrm{I} \xrightarrow{\beta} P_1P_2P_3P_4 \equiv (\lambda xy.y)P_2P_3y_2 \xrightarrow{\beta} P_3y_2 = y_3 \iff P_3 = \lambda t.y_3$$

and finally

$$\Delta \mathrm{S} \xrightarrow{\beta} P_1P_3(P_2P_3)P_4 \equiv (\lambda xy.y)(\lambda t.y_3)(P_2(\lambda t.y_3))y_2 \iff P_2 = \lambda ab.y_1$$