1

UNTYPED λ-CALCULUS

When are two functions equal? Classically in mathematics, *functions are graphs*. A function from a domain to a codomain, $f : X \longrightarrow Y$, is seen as a subset of the product space: $f \subset X \times Y$. Any two functions are identical if they map equal inputs to equal outputs; and a function it is completely determined by what its outputs are. This vision is called ***extensional***.

From a computational point of view, this perspective could seem incomplete in some cases. Computationally, we usually care not only about the result but, crucially, about *how* it can be computed. Classically in computer science, *functions are formulae*; and two functions mapping equal inputs to equal outputs need not to be equal. For instance, two sorting algorithms can have a different efficiency or different memory requisites, even if they output the same sorted list. This vision, where two functions are equal if and only if they are given by essentially the same formula is called ***intensional***.

The **λ-calculus** is a collection of formal systems, all of them based on the lambda notation introduced by Alonzo Church in the 1930s while trying to develop a foundational notion of functions (*as formulae*) on mathematics. It is a logical theory of functions, where application and abstraction are primitive notions; and, at the same time, it is also one of the simplest programming languages, in which many other full-fledged languages are based, as we will explain in detail later.

The **untyped** or **pure λ-calculus** is, syntactically, the simplest of those formal systems. In it, a function does not need a domain nor a codomain; every function is a formula that can be directly applied to any expression. It even allows functions to be applied to themselves, a notion that would be troublesome in our usual set-theoretical foundations. In particular, if f is a member of its own domain, the infinite descending sequence

$$f \ni \{f, f(f)\} \ni f \ni \{f, f(f)\} \ni \dots,$$

would exist, thus contradicting the **regularity axiom** of Zermelo-Fraenkel set theory (see, for example, [Kun11]). However, untyped λ-calculus presents some problems such as non-terminating functions.

This presentation of the untyped lambda calculus will follow [HS08] and [Sel13].

1.1 THE UNTYPED λ-CALCULUS

As a formal language, the untyped λ-calculus is given by a set of equations between expressions called λ-*terms*, and equivalences between them can be computed using some manipulation rules. These λ-terms can stand for functions or arguments indistinctly: they all use the same λ-notation in order to define function abstractions and applications.

The λ-**notation** allows a function to be written and inlined as any other element of the language, identifying it with the formula it represents and admitting a more compact notation. For example, the polynomial function $p(x) = x^2 + x$ would be written in λ-calculus as $\lambda x.\ x^2 + x$; and $p(2)$ would be written as $(\lambda x.\ x^2 + x)(2)$. In general, $\lambda x.M$ is a function taking x as an argument and returning M, which is a term where x may appear in.

The use of λ-notation also eases the writing of **higher-order functions**, functions whose arguments or outputs are functions themselves. For instance,

$$\lambda f.(\ \lambda y.\ f(f(y))\ )$$

would be a function taking f as an argument and returning $\lambda y.\ f(f(y))$, which is itself a function; most commonly written as $f \circ f$. In particular,

$$\Big(\big(\lambda f.(\ \lambda y.\ f(f(y))\ )\big)\big(\lambda x.\ x^2 + x\big)\Big)(1)$$

evaluates to 6.

Definition 1 (Lambda terms)**.** λ**-terms** are constructed inductively using the following rules

- every ***variable***, taken from an infinite countable set of variables and usually written as lowercase single letters ($x, y, z, \dots$), is a λ-term;
- given two λ-terms M, N; its ***application***, MN, is a λ-term;
- given a λ-term M and a variable x, its ***abstraction***, $\lambda x.M$, is a λ-term;
- every possible λ-term can be constructed using these rules, no other λ-term exists.

Equivalently, they can also be defined by the following Backus-Naur form,

$$\texttt{Term} ::= x \mid (\texttt{Term}\ \texttt{Term}) \mid (\lambda x.\texttt{Term}) \quad ,$$

where x can be any variable.

By convention, we omit outermost parentheses and assume left-associativity, for example, MNP will always mean $(MN)P$. Note that the application of λ-terms is different from its composition, which is distributive and will be formally defined later. We consider λ-abstraction as having the lowest precedence. For example, $\lambda x.MN$ should be read as $\lambda x.(MN)$ instead of $(\lambda x.M)N$.

1.2 FREE AND BOUND VARIABLES, SUBSTITUTION

In λ-calculus, the scope of a variable restricts to the λ-abstraction where it appeared, if any. Thus, the same variable can be used multiple times on the same term independently. For example, in $(\lambda x.x)(\lambda x.x)$, the variable x appears twice with two different meanings.

Any ocurrence of a variable x inside the *scope* of a lambda is said to be ***bound***; and any variable without bound ocurrences is said to be ***free***. Formally, we can define the set of free variables on a given term as follows.

Definition 2 (Free variables)**.** The **set of free variables** of a term M is defined inductively as

$$\begin{aligned} \mathrm{FV}(x) &= \{x\}, \\ \mathrm{FV}(MN) &= \mathrm{FV}(M) \cup \mathrm{FV}(N), \\ \mathrm{FV}(\lambda x.M) &= \mathrm{FV}(M) \setminus \{x\}. \end{aligned}$$

Evaluation in λ-calculus relies in the notion of ***substitution***. Any free ocurrence of a variable can be substituted by a term, as we do when we are evaluating terms. For instance, in the previous example, we evaluated $(\lambda x.\ x^2 + x)(2)$ by substituting 2 in the place of x inside $x^2 + x$; as in

$$(\lambda x.\ x^2 + x)(2) \xrightarrow{x \mapsto 2} 2^2 + 2.$$

This, however, should be done avoiding the unintended binding which happens when a variable is substituted inside the scope of a binder with the same name, as in the following example: if we were to evaluate the expression $(\lambda x.\ yx)(\lambda z.\ xz)$, where x appears two times (once bound and once free), we should substitute y by $(\lambda z.xz)$ on $(\lambda x.yx)$ and x (the free variable) would get tied to x (the bounded variable)

$$(\lambda y.\lambda x.yx)(\lambda z.\ xz) \xrightarrow{y \mapsto (\lambda z.xz)} (\lambda x.(\lambda z.xz)x).$$

To avoid this, the bounded x must be given a new name before the substitution, which must be carried as

$$(\lambda y.\lambda u.yu)(\lambda z.\ xz) \xrightarrow{y \mapsto (\lambda z.xz)} (\lambda u.(\lambda z.xz)u),$$

keeping the free character of x.

Definition 3 (Substitution on lambda terms)**.** The **substitution** of a variable x by a term N on M is written as $M[N/x]$ and is defined inductively as

$$\begin{aligned} x[N/x] &\equiv N, \\ y[N/x] &\equiv y, && \text{if } y \neq x, \\ (MP)[N/x] &\equiv (M[N/x])(P[N/x]), \\ (\lambda x.P)[N/x] &\equiv \lambda x.P, \\ (\lambda y.P)[N/x] &\equiv \lambda y.P[N/x] && \text{if } y \notin \mathrm{FV}(N), \\ (\lambda y.P)[N/x] &\equiv \lambda z.P[z/y][N/x] && \text{if } y \in \mathrm{FV}(N), \end{aligned}$$

where, in the last clause, z is a fresh unused variable.

We could define a criterion for choosing exactly what this new variable should be, or simply accept that our definition will not be exactly well-defined, but only *well-defined up to a change on the name of the variables*. This equivalence relation will be defined formally on the next section. In practice, it is common to follow the *Barendregt's variable convention*, which simply assumes that bound variables have been renamed to be distinct.

1.3 α-EQUIVALENCE

In λ-terms, variables are only placeholders and its name, as we have seen before, is not relevant. Two λ-terms whose only difference is the naming of the variables are called α-equivalent. For example,

$$(\lambda x.\lambda y.x\ y) \quad \text{is } \alpha\text{-equivalent to} \quad (\lambda f.\lambda x.f\ x).$$

α-equivalence formally captures the fact that the name of a bound variable can be changed without changing the meaning of the term. This idea appears recurrently on mathematics; for example, the renaming of variables of integration or the variable on a limit are a examples of α-equivalence.

$$\int_0^1 x^2\ dx = \int_0^1 y^2\ dy; \qquad \lim_{x\to\infty} \frac{1}{x} = \lim_{y\to\infty} \frac{1}{y}.$$

Definition 4 ($\alpha - equivalence$). **α-equivalence** is the smallest relation $=_\alpha$ on λ-terms that is an equivalence relation, that is to say that

- it is *reflexive*, $M =_\alpha M$;
- it is *symmetric*, if $M =_\alpha N$, then $N =_\alpha M$;
- and it is *transitive*, if $M =_\alpha N$ and $N =_\alpha P$, then $M =_\alpha P$;

and it is compatible with the structure of lambda terms,

- if $M =_\alpha M'$ and $N =_\alpha N'$, then $MN =_\alpha M'N'$;
- if $M =_\alpha M'$, then $\lambda x.M =_\alpha \lambda x.M'$;
- if y does not appear on M, $\lambda x.M =_\alpha \lambda y.M[y/x]$.

1.4 β-REDUCTION

The core idea of evaluation in λ-calculus is captured by the notion of **β-reduction**. Until now, evaluation has been only informally described; it is time to define it as a relation, $\twoheadrightarrow_\beta$, going from the initial term to any of its partial evaluations. We will firstly consider a *one-step reduction* relationship, called $\rightarrow_\beta$, which will be extended by transitivity to $\twoheadrightarrow_\beta$.

Ideally, we would like to define evaluation as a series of reductions into a canonical form which could not be further reduced. Unfortunately, as we will see later, it is not possible to find, in general, that canonical form.

Definition 5 (β – *reduction*). The **single-step β-reduction** is the smallest relation on λ-terms capturing the notion of evaluation and preserving the structure of λ-abstractions and applications. That is, the smallest relation containing

- $(\lambda x.M)N \longrightarrow_\beta M[N/x]$ for any terms M, N and any variable x,
- $MN \longrightarrow_\beta M'N$ and $NM \longrightarrow_\beta NM'$ for any M, M' such that $M \longrightarrow_\beta M'$, and
- $\lambda x.M \longrightarrow_\beta \lambda x.M'$, for any M, M' such that $M \longrightarrow_\beta M'$.

The reflexive transitive closure of $\longrightarrow_\beta$ is written as $\twoheadrightarrow_\beta$. The symmetric closure of $\twoheadrightarrow_\beta$ is called **β-equivalence** and written as $=_\beta$ or simply $=$.

1.5 η-REDUCTION

Although we lost the extensional view of functions when we decided to adopt the *functions as formulae* perspective, the idea of function extensionality in λ-calculus can be partially recovered by the notion of η-reduction. This form of *function extensionality for λ-terms* can be captured by the notion that any term which simply applies a function to the argument it takes can be reduced to the actual function. That is, any $\lambda x.Mx$ can be reduced to M.

Definition 6 (η – *reduction*). The **η-reduction** is the smallest relation on λ-terms satisfiying the same congruence rules as β-reduction and the following axiom

$$\lambda x.Mx \longrightarrow_\eta M, \text{ for any } x \notin \mathrm{FV}(M).$$

We define single-step $\beta\eta$-reduction as the union of β-reduction and η-reduction. This will be written as $\longrightarrow_{\beta\eta}$, and its reflexive transitive closure will be $\twoheadrightarrow_{\beta\eta}$.

Note that, in the particular case where M is itself a λ-abstraction, η-reduction is simply a particular case of β-reduction.

1.6 CONFLUENCE

As we mentioned above, it is not possible in general to evaluate a λ-term into a canonical, non-reducible term. However, we will be able to prove that, in the cases where it exists, it is unique. This property is a consequence of a sightly more general one, ***confluence***, which can be defined in any abstract rewriting system.

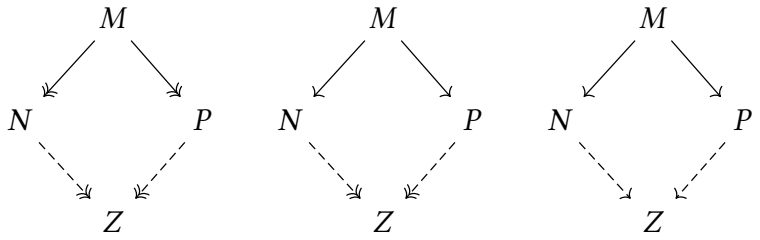
Definition 7 (Confluence)**.** A relation $\longrightarrow$ on a set S is **confluent** if, given its reflexive transitive closure $\twoheadrightarrow$, for any $M, N, P \in S$, $M \twoheadrightarrow N$ and $M \twoheadrightarrow P$ imply the existence of some $Z \in S$ such that $N \twoheadrightarrow Z$ and $P \twoheadrightarrow Z$.

Given any binary relation $\longrightarrow$ of which $\twoheadrightarrow$ is its reflexive transitive closure, we can consider three seemingly related properties

- the **confluence** (also called *Church-Rosser property*) we have just defined,

- the **quasidiamond property**, which assumes $M \longrightarrow N$ and $M \longrightarrow P$,
- the **diamond property**, which is defined substituting $\twoheadrightarrow$ by $\longrightarrow$ on the definition on confluence.

Diagrammatically, the three properties can be represented as



and we can show that the diamond relation implies confluence; while the quasidiamond does not. Both claims are easy to prove, and they show us that, in order to prove confluence for a given relation, we can use the diamond property instead of the quasidiamond property.

The statement of $\twoheadrightarrow_\beta$ and $\twoheadrightarrow_{\beta\eta}$ being confluent is what we call the ***Church-Rosser Theorem***. The definition of a relation satisfying the diamond property and whose reflexive transitive closure is $\twoheadrightarrow_{\beta\eta}$ will be the core of our proof.

1.7 THE CHURCH-ROSSER THEOREM

The proof presented here is due to Tait and Per Martin-Löf; an earlier but more convoluted proof was discovered by Alonzo Church and Barkley Rosser in 1935 (see [Bar84] and [Pol95]). It is based on the idea of parallel one-step reduction.

Definition 8 (Parallel one-step reduction)**.** We define the **parallel one-step reduction** relation on λ-terms, $\triangleright$, as the smallest relation satisfying that the following properties

- reflexivity, $x \triangleright x$;
- parallel application, $PN \triangleright P'N'$;
- congruence to λ-abstraction, $\lambda x.N \triangleright \lambda x.N'$;
- parallel substitution, $(\lambda x.P)N \triangleright P'[N'/x]$;
- and extensionality, $\lambda x.Px \triangleright P'$, if $x \notin \mathrm{FV}(P)$,

hold for any variable x and any terms N, N', P, P' such that $P \triangleright P'$ and $N \triangleright N'$.

Using the first three rules, it is trivial to show that this relation is in fact reflexive.

Lemma 1. *The reflexive transitive closure of $\triangleright$ is $\twoheadrightarrow_{\beta\eta}$. In particular, given any λ-terms M, M',*

1. *if $M \longrightarrow_{\beta\eta} M'$, then $M \triangleright M'$.*
2. *if $M \triangleright M'$, then $M \twoheadrightarrow_{\beta\eta} M'$;*

Proof. 1. We can prove this by exhaustion and structural induction on λ-terms, the possible ways in which we arrive at $M \longrightarrow M'$ are

- $(\lambda x.M)N \longrightarrow M[N/x]$; where we know that, by parallel substitution and reflexivity $(\lambda x.M)N \triangleright M[N/x]$;
- $MN \longrightarrow M'N$ and $NM \longrightarrow NM'$; where we know that, by induction $M \triangleright M'$, and by parallel application and reflexivity, $MN \triangleright M'N$ and $NM \triangleright NM'$;
- congruence to λ-abstraction, which is a shared property between the two relations where we can apply structural induction again;
- $\lambda x.Px \longrightarrow P$, where $x \notin \mathrm{FV}(P)$ and we can apply extensionality for $\triangleright$ and reflexivity.

2. We can prove this by induction on any derivation of $M \triangleright M'$. The possible ways in which we arrive at this are

- the trivial one, reflexivity;
- parallel application $NP \triangleright N'P'$, where, by induction, we have $P \twoheadrightarrow P'$ and $N \twoheadrightarrow N'$. Using two steps, $NP \twoheadrightarrow N'P \twoheadrightarrow N'P'$ we prove $NP \twoheadrightarrow N'P'$;
- congruence to λ-abstraction $\lambda x.N \triangleright \lambda x.N'$, where, by induction, we know that $N \twoheadrightarrow N'$, so $\lambda x.N \twoheadrightarrow \lambda x.N'$;
- parallel substitution, $(\lambda x.P)N \triangleright P'[N'/x]$, where, by induction, we know that $P \twoheadrightarrow P'$ and $N \twoheadrightarrow N'$. Using multiple steps, $(\lambda x.P)N \twoheadrightarrow (\lambda x.P')N \twoheadrightarrow (\lambda x.P')N' \longrightarrow P'[N'/x]$;
- extensionality, $\lambda x.Px \triangleright P'$, where by induction $P \twoheadrightarrow P'$, and trivially, $\lambda x.Px \twoheadrightarrow \lambda x.P'x$.

Because of this, the reflexive transitive closure of $\triangleright$ should be a subset and a superset of $\twoheadrightarrow$ at the same time. □

Lemma 2 (Substitution Lemma)**.** *Assuming $M \triangleright M'$ and $U \triangleright U'$, $M[U/y] \triangleright M'[U'/y]$.*

Proof. We apply structural induction on derivations of $M \triangleright M'$, depending on what the last rule we used to derive it was.

- Reflexivity, $M = x$. If $x = y$, we simply use $U \triangleright U'$; if $x \neq y$, we use reflexivity on x to get $x \triangleright x$.
- Parallel application. By induction hypothesis, $P[U/y] \triangleright P'[U'/y]$ and $N[U/y] \triangleright N'[U'/y]$, hence $(PN)[U/y] \triangleright (P'N')[U'/y]$.
- Congruence. By induction, $N[U/y] \triangleright N'[U'/y]$ and $\lambda x.N[U/y] \triangleright \lambda x.N'[U'/y]$.
- Parallel substitution. By induction, $P[U/y] \triangleright P'[U'/y]$ and $N[U/y] \triangleright N[U'/y]$, hence $((\lambda x.P)N)[U/y] \triangleright P'[U'/y][N'[U'/y]/x] = P'[N'/x][U'/y]$.
- Extensionality, given $x \notin \mathrm{FV}(P)$. By induction, $P \triangleright P'$, hence $\lambda x.P[U/y]x \triangleright P'[U'/y]$.

Note that we are implicitely assuming the Barendregt's variable convention; all variables have been renamed to avoid clashes. □

Definition 9 (Maximal parallel one-step reduct)**.** The **maximal parallel one-step reduct** M^* of a λ-term M is defined inductively as

- $x^* = x$, if x is a variable;

- $(PN)^* = P^*N^*$;
- $((\lambda x.P)N)^* = P^*[N^*/x]$;
- $(\lambda x.N)^* = \lambda x.N^*$;
- $(\lambda x.Px)^* = P^*$, given $x \notin \mathrm{FV}(P)$.

Lemma 3 (Diamond property of parallel reduction)**.** *Given any M' such that $M \triangleright M'$, $M' \triangleright M^*$. Parallel one-step reduction has the diamond property.*

Proof. We apply again structural induction on the derivation of $M \triangleright M'$.

- Reflexivity gives us $M' = x = M^*$.
- Parallel application. By induction, we have $P \triangleright P^*$ and $N \triangleright N^*$; depending on the form of P, we have
  - P is not a λ-abstraction and $P'N' \triangleright P^*N^* = (PN)^*$.
  - $P = \lambda x.Q$ and $P \triangleright P'$ could be derived using congruence to λ-abstraction or extensionality. On the first case we know by induction hypothesis that $Q' \triangleright Q^*$ and $(\lambda x.Q')N' \triangleright Q^*[N^*/x]$. On the second case, we can take $P = \lambda x.Rx$, where, $R \triangleright R'$. By induction, $(R'x) \triangleright (Rx)^*$ and now we apply the substitution lemma to have $R'N' = (R'x)[N'/x] \triangleright (Rx)^*[N^*/x]$.
- Congruence. Given $N \triangleright N'$; by induction $N' \triangleright N^*$, and depending on the form of N we have two cases
  - N is not of the form Px where $x \notin \mathrm{FV}(P)$; we can apply congruence to λ-abstraction.
  - $N = Px$ where $x \notin \mathrm{FV}(P)$; and $N \triangleright N'$ could be derived by parallel application or parallel substitution. On the first case, given $P \triangleright P'$, we know that $P' \triangleright P^*$ by induction hypothesis and $\lambda x.P'x \triangleright P^*$ by extensionality. On the second case, $N = (\lambda y.Q)x$ and $N' = Q'[x/y]$, where $Q \triangleright Q'$. Hence $P \triangleright \lambda y.Q'$, and by induction hypothesis, $\lambda y.Q' \triangleright P^*$.
- Parallel substitution, with $N \triangleright N'$ and $Q \triangleright Q'$; we know that $M^* = Q^*[N^*/x]$ and we can apply the substitution lemma (lemma 2) to get $M' \triangleright M^*$.
- Extensionality. We know that $P \triangleright P'$ and $x \notin \mathrm{FV}(P)$, so by induction hypothesis we know that $P' \triangleright P^* = M^*$. □

Theorem 1 (Church-Rosser Theorem)**.** *The relation $\twoheadrightarrow_{\beta\eta}$ is confluent.*

Proof. Parallel reduction, $\triangleright$, satisfies the diamond property (lemma 3), which implies the Church-Rosser property. Its reflexive transitive closure is $\twoheadrightarrow_{\beta\eta}$ (lemma 1), whose diamond property implies confluence for $\longrightarrow_{\beta\eta}$. □

1.8 NORMALIZATION

Once the Church-Rosser theorem is proved, we can formally define the notion of a normal form as a completely reduced λ-term.

Definition 10 (Normal forms)**.** A λ-term is said to be in **β-normal form** if β-reduction cannot be applied to it or any of its subformulas. We define **η-normal forms** and **$\beta\eta$-normal forms** analogously.

Fully evaluating λ-terms usually means to apply reductions to them until a normal form is reached. We know, by virtue of Theorem 1, that, if a normal form for a particular term exists, it is unique; but we do not know whether a normal form actually exists. We say that a term **has** a normal form when it can be reduced to a normal form.

Definition 11. A term is **weakly normalizing** if there exists a sequence of reductions from it to a normal form. It is **strongly** normalizing if every sequence of reductions is finite.

A consequence of Theorem 1 is that a weakly normalizing term has a unique normal form. Strong normalization implies weak normalization, but the converse is not true; as an example, the term

$$\Omega = (\lambda x.(xx))(\lambda x.(xx))$$

is neither weakly nor strongly normalizing; and the term $(\lambda x.\lambda y.y)\ \Omega\ (\lambda x.x)$ is weakly but not strongly normalizing. It can be reduced to a normal form as

$$(\lambda x.\lambda y.y)\ \Omega\ (\lambda x.x) \longrightarrow_\beta (\lambda x.x).$$

1.9 STANDARIZATION AND EVALUATION STRATEGIES

We would like to find a β-reduction strategy such that, if a term has a normal form, it can be found by following that strategy. Our basic result will be the **standarization theorem**, which shows that, if a β-reduction to a normal form exists, then a sequence of β-reductions from left to right on the λ-expression will be able to find it. From this result, we will be able to prove that the reduction strategy that always reduces the leftmost β-abstraction will always find a normal form if it exists.

This section follows [Kas00], [Bar94] and [Bar84].

Definition 12 (Leftmost one-step reduction)**.** We define the relation $M \longrightarrow_n N$ when N can be obtained by β-reducing the n-th leftmost β-reducible application of the expression. We call $\longrightarrow_1$ the **leftmost one-step reduction** and we write it as $\longrightarrow_l$; accordingly, $\twoheadrightarrow_l$ is its reflexive transitive closure.

Definition 13 (Standard sequence)**.** A sequence of β-reductions $M_0 \longrightarrow_{n_1} M_1 \longrightarrow_{n_2} M_2 \longrightarrow_{n_3} \cdots \longrightarrow_{n_k} M_k$ is **standard** if $\{n_i\}$ is a non-decreasing sequence.

We will prove that every term that can be reduced to a normal form can be reduced to it using a standard sequence, from this result, the existence of an optimal beta reduction strategy, in the sense that it will always reach a normal form if one exists, will follow as a corollary.

Theorem 2 (Standarization theorem)**.** *If $M \twoheadrightarrow_\beta N$, there exists a standard sequence from M to N.*

Proof. We start by defining the following two binary relations. The first one is the minimal reflexive transitive relation on λ-terms capturing a form of β-reduction called *head β-reduction*; that is, it is the minimal relation $\twoheadrightarrow_h$ such that

- $A \twoheadrightarrow_h A$,
- $(\lambda x.A_0)A_1A_2 \dots A_m \twoheadrightarrow_h A_0[A_1/x]A_2 \dots A_m$, for any term of the form $A_1A_2 \dots A_n$, and
- $A \twoheadrightarrow_h C$ for any terms A, B, C such that $A \twoheadrightarrow_h B \twoheadrightarrow_h C$.

The second one is called *standard reduction*. It is the minimal relation between λ-terms such that

- $M \twoheadrightarrow_h x$ implies $M \twoheadrightarrow_s x$, for any variable x,
- $M \twoheadrightarrow_h AB$, $A \twoheadrightarrow_s C$ and $B \twoheadrightarrow_s D$, imply $M \twoheadrightarrow_s CD$,
- $M \twoheadrightarrow_h \lambda x.A$ and $A \twoheadrightarrow_s B$ imply $M \longrightarrow_s \lambda x.B$.

We can check the following trivial properties by structural induction

1. $\twoheadrightarrow_h$ implies $\twoheadrightarrow_l$,
2. $\twoheadrightarrow_s$ implies the existence of a standard β-reduction,
3. $\twoheadrightarrow_s$ is reflexive, by induction on the structure of a term,
4. if $M \twoheadrightarrow_h N$, then $MP \twoheadrightarrow_h NP$,
5. if $M \twoheadrightarrow_h N \twoheadrightarrow_s P$, then $M \twoheadrightarrow_s P$,
6. if $M \twoheadrightarrow_h N$, then $M[P/x] \twoheadrightarrow_h N[P/x]$,
7. if $M \twoheadrightarrow_s N$ and $P \twoheadrightarrow_s Q$, then $M[P/z] \twoheadrightarrow_s N[Q/z]$.

And now we can prove that $K \twoheadrightarrow_s (\lambda x.M)N$ implies $K \twoheadrightarrow_s M[N/x]$. From the fact that $K \twoheadrightarrow_s (\lambda x.M)N$, we know that there must exist P and Q such that $K \twoheadrightarrow_h PQ$, $P \twoheadrightarrow_s \lambda x.M$ and $Q \twoheadrightarrow_s N$; and from $P \twoheadrightarrow_s \lambda x.M$, we know that there exists W such that $P \twoheadrightarrow_h \lambda x.W$ and $W \twoheadrightarrow_s M$. From all this information, we can conclude that

$$K \twoheadrightarrow_h PQ \twoheadrightarrow_h (\lambda x.W)Q \twoheadrightarrow W[Q/x] \twoheadrightarrow_s M[N/x];$$

which, by (3.), implies $K \twoheadrightarrow_s M[N/x]$.

We finally prove that, if $K \twoheadrightarrow_s M \longrightarrow_\beta N$, then $K \twoheadrightarrow_s N$. This proves the theorem, as every β-reduction $M \twoheadrightarrow_s M \twoheadrightarrow_\beta N$ implies $M \twoheadrightarrow_s N$. We analize the possible ways in which $M \longrightarrow_\beta N$ can be derived.

1. If $K \twoheadrightarrow_s (\lambda x.M)N \longrightarrow_\beta M[N/x]$, it has been already showed that $K \twoheadrightarrow_s M[N/x]$.
2. If $K \twoheadrightarrow_s MN \longrightarrow_\beta M'N$ with $M \longrightarrow_\beta M'$, we know that there exist $K \twoheadrightarrow_h WQ$ such that $W \twoheadrightarrow_s M$ and $Q \twoheadrightarrow_s N$; by induction $W \twoheadrightarrow_s M'$, and then $WQ \twoheadrightarrow_s M'N$. The case $K \twoheadrightarrow_s MN \longrightarrow_\beta MN'$ is entirely analogous.
3. If $K \twoheadrightarrow_s \lambda x.M \longrightarrow_\beta \lambda x.M'$, with $M \longrightarrow_\beta M'$, we know that there exists W such that $K \twoheadrightarrow_h \lambda x.W$ and $W \twoheadrightarrow_s M$. By induction $W \twoheadrightarrow_s M'$, and $K \twoheadrightarrow_s \lambda x.M'$. □

Corollary 1 (Leftmost reduction theorem)**.** *We define the **leftmost reduction strategy** as the strategy that reduces the leftmost β-reducible application at each step. If M has a normal form, the leftmost reduction strategy will lead to it.*

Proof. Note that, if $M \to_n N$, where N is in β-normal form; n must be exactly 1. If M has a normal form and $M \twoheadrightarrow_\beta N$, by Theorem 2, there must exist a standard sequence from M to N whose last step is of the form $\to_l$; as the sequence is non-decreasing, every step has to be of the form $\to_l$. □

1.10 SKI COMBINATORS

As we have seen in previous sections, untyped λ-calculus is already a very syntactically simple system; but it can be further reduced to a few λ-terms without losing its expressiveness. In particular, untyped λ-calculus can be *essentially* recovered from only two of its terms; these are

- $S = \lambda x.\lambda y.\lambda z.xz(yz)$, and
- $K = \lambda x.\lambda y.x$.

A language can be defined with these combinators and function application. Every λ-term can be translated to this language and recovered up to $=_{\beta\eta}$ equivalence. For example, the identity λ-term, I, can be written as

$$I = \lambda x.x = SKK.$$

It is common to also add the $I = \lambda x.x$ as a basic term to this language, even if it can be written in terms of S and K, as a way to ease the writing of long complex terms. Terms written with these combinators are called ***SKI-terms***.

The language of **SKI-terms** can be defined by the following Backus-Naus form

$$\texttt{SKI} ::= x \mid (\texttt{SKI}\ \texttt{SKI}) \mid S \mid K \mid I \quad ,$$

where x are free variables.

Definition 14 (Lambda transform)**.** The **Lambda-transform** of a SKI-term is a λ-term defined recursively as

- $\mathfrak{L}(x) = x$, for any variable x;
- $\mathfrak{L}(I) = (\lambda x.x)$;
- $\mathfrak{L}(K) = (\lambda x.\lambda y.x)$;
- $\mathfrak{L}(S) = (\lambda x.\lambda y.\lambda z.xz(yz))$;
- $\mathfrak{L}(X\,Y) = \mathfrak{L}(X)\mathfrak{L}(Y)$.

Definition 15 (Bracket abstraction)**.** The **bracket abstraction** of the SKI-term U on the variable x is written as $[x].U$ and defined recursively as

- $[x].x = I$;
- $[x].M = KM$, if $x \notin \mathrm{FV}(M)$;
- $[x].Ux = U$, if $x \notin \mathrm{FV}(U)$;
- $[x].U\,V = S([x].U)([x].V)$, otherwise.

where FV is the set of free variables; as defined on Definition 2.

Definition 16 (SKI abstraction)**.** The **SKI abstraction** of a λ-term M, written as $\mathfrak{H}(M)$ is defined recursively as

- $\mathfrak{H}(x) = x$, for any variable x;
- $\mathfrak{H}(MN) = \mathfrak{H}(M)\mathfrak{H}(N)$;
- $\mathfrak{H}(\lambda x.M) = [x].\mathfrak{H}(M)$;

where $[x].U$ is the bracket abstraction of the SKI-term U.

Theorem 3 (SKI combinators and lambda terms)**.** *The SKI-abstraction is a retraction of the Lambda-transform of the term, that is, for any SKI-term U,*

$$\mathfrak{H}(\mathfrak{L}(U)) = U.$$

Proof. By structural induction on U,

- $\mathfrak{H}\mathfrak{L}(x) = x$, for any variable x;
- $\mathfrak{H}\mathfrak{L}(I) = [x].x = I$;
- $\mathfrak{H}\mathfrak{L}(K) = [x].[y].x = [x].Kx = K$;
- $\mathfrak{H}\mathfrak{L}(S) = [x].[y].[z].xz(yz) = [x].[y].Sxy = S$; and
- $\mathfrak{H}\mathfrak{L}(MN) = MN$. □

In general this translation is not an isomorphism. As an example

$$\mathfrak{L}(\mathfrak{H}(\lambda u.vu)) = \mathfrak{L}(v) = v.$$

However, the λ-terms can be essentially recovered if we relax equality between λ-terms to mean $=_{\beta\eta}$.

Theorem 4 (Recovering lambda terms from SKI combinators)**.** *For any λ-term M,*

$$\mathfrak{L}(\mathfrak{H}(M)) =_{\beta\eta} M.$$

Proof. We can firstly prove by structural induction that $\mathfrak{L}([x].M) = \lambda x.\mathfrak{L}(M)$ for any M. In fact, we know that $\mathfrak{L}([x].x) = \lambda x.x$ for any variable x; we also know that

$$\begin{aligned}\mathfrak{L}([x].MN) &= \mathfrak{L}(S([x].M)([x].N)) \\ &= (\lambda x.\lambda y.\lambda z.xz(yz))(\lambda x.\mathfrak{L}(M))(\lambda x.\mathfrak{L}(N)) \\ &= \lambda z.\mathfrak{L}(M)\mathfrak{L}(N);\end{aligned}$$

also, if x is free in M,

$$\mathfrak{L}([x].M) = \mathfrak{L}(KM) = (\lambda x.\lambda y.x)\mathfrak{L}(M) =_{\beta} \lambda x.\mathfrak{L}(M);$$

and finally, if x is free in U,

$$\mathfrak{L}([x].Ux) = \mathfrak{L}(U) =_{\eta} \lambda x.\mathfrak{L}(U)x\ .$$

Now we can use this result to prove the main theorem. Again by structural induction,

- $\mathfrak{L}\mathfrak{H}(x) = x$;
- $\mathfrak{L}\mathfrak{H}(MN) = \mathfrak{L}\mathfrak{H}(M)\mathfrak{L}\mathfrak{H}(N) = MN$;
- $\mathfrak{L}\mathfrak{H}(\lambda x.M) = \mathfrak{L}([x].\mathfrak{H}(M)) =_{\beta\eta} \lambda x.\mathfrak{L}\mathfrak{H}(M) = \lambda x.M$. □

1.11 TURING COMPLETENESS

Three different notions of computability were proposed in the 1930s

- the **general recursive functions** were defined by Herbrand and Gödel. They form a class of functions over the natural numbers closed under composition, recursion and unbound search.
- the λ-**definable functions** were proposed by Church. They are functions on the natural numbers that can be represented by λ-terms.
- the **Turing computable functions**, proposed by Alan Turing as the functions that can be defined on a theoretical model of a machine, the *Turing machines*.

In [Chu36] and [Tur37], Church and Turing proved the equivalence of the three definitions. This lead to the metatheoretical ***Church-Turing thesis***, which postulated the equivalence between these models of computation and the intuitive notion of *effective calculability* mathematicians were using. In practice, this means that the λ-calculus, as a programming language, is as expressive as Turing machines; it can define every computable function. It is Turing-complete.

A complete implementation of untyped λ-calculus is discussed in the chapter on Mikrokosmos; and a detailed description on how to use the untyped λ-calculus as a programming language is given in the chapter "".

2

SIMPLY TYPED λ-CALCULUS

Types were introduced in mathematics as a response to the Russell's paradox, found in the first naive axiomatizations of set theory. An attempt to use untyped λ-calculus as a foundational logical system by Church suffered from the ***Rosser-Kleene paradox***, as detailed in [KR35] and [Cur46]; and types were a way to avoid it. Once types are added, a deep connection between λ-calculus and logic arises. This connection will be discussed in the next chapter.

In programming languages, types indicate how the programmer intends to use the data, prevent errors and enforce certain invariants and levels of abstraction in programs. The role of types in λ-calculus when interpreted as a programming language closely matches what we would expect of types in any common programming language, and typed λ-calculus has been the basis of many modern type systems for programming languages.

Simply typed λ-calculus is a refinment of the untyped λ-calculus. In it, each term has a type, which limits how it can be combined with other terms. Only a set of basic types and function types between any to types are considered in this system. Whereas functions in untyped λ-calculus could be applied over any term, now a function of type $A \longrightarrow B$ can only be applied over a term of type A, to produce a new term of type B, where A and B could be, themselves, function types.

We will give now a presentation of simply typed λ-calculus based on [HS08]. Our presentation will rely only on the *arrow type constructor* $\longrightarrow$. While other presentations of simply typed λ-calculus extend this definition with type constructors providing pairs or union types, as it is done in [Sel13], it seems clearer to present a first minimal version of the λ-calculus. Such extensions will be explained later, and its exposition will profit from the logical interpretation that we will explain in "propositions as types".

2.1 SIMPLE TYPES

We start assuming a set of **basic types**. Those basic types would correspond, in a programming language interpretation, with the fundamental types of the language. Examples would be the type of strings or the type of integers. Minimal presentations of λ-calculus tend to use only one basic type.