Category Theory and Lambda Calculus

Mario Román

November 30, 2017

CONTENTS

ABSTRACT

This is the abstract.

Part I

LAMBDA CALCULUS

The λ-calculus is a collection of systems formalizing the notion of functions. They can be seen as programming languages and formal logics at the same time. We focus on the properties of the untyped λ-calculus and simply typed λ-calculus and its relation to logic.

1

UNTYPED λ-CALCULUS

When are two functions equal? Classically in mathematics, *functions are graphs*. A function from a domain to a codomain, $f\colon X \to Y$, 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,

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

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 $\longrightarrow_\beta$, 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 ($\beta - reduction$). 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 ($\eta - reduction$). 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 $\to$ on a set $\mathcal{S}$ is **confluent** if, given its reflexive transitive closure $\twoheadrightarrow$, for any $M, N, P \in \mathcal{S}$, $M \twoheadrightarrow N$ and $M \twoheadrightarrow P$ imply the existence of some $Z \in \mathcal{S}$ such that $N \twoheadrightarrow Z$ and $P \twoheadrightarrow Z$.

Given any binary relation $\to$ 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 $M \to N$ and $M \to P$,
- the **diamond property**, which is defined substituting $\twoheadrightarrow$ by $\to$ on the definition on confluence.

Diagrammatically, the three properties can be represented as

$$\begin{tikzcd}
& M \arrow[dl, two heads] \arrow[dr, two heads] & & & M \arrow[dl] \arrow[dr] & & & M \arrow[dl] \arrow[dr] & \\
N \arrow[dr, two heads, dashed] & & P \arrow[dl, two heads, dashed] & N \arrow[dr, two heads, dashed] & & P \arrow[dl, two heads, dashed] & N \arrow[dr, dashed] & & P \arrow[dl, dashed] \\
& Z & & & Z & & & Z &
\end{tikzcd}$$

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, $\rhd$, as the smallest relation satisfying that the following properties

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

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

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

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

1. *if* $M \to_{\beta\eta} M'$*, then* $M \rhd M'$*.*
2. *if* $M \rhd M'$*, 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 $M \to M'$ are

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

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

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

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

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

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

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

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 $M \rhd M'$, $M' \rhd M^*$. Parallel one-step reduction has the diamond property.*

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

- Reflexivity gives us $M' = x = M^*$.
- Parallel application. By induction, we have $P \rhd P^*$ and $N \rhd N^*$; depending on the form of P, we have
  - P is not a λ-abstraction and $P'N' \rhd P^*N^* = (PN)^*$.
  - $P = \lambda x.Q$ and $P \rhd P'$ could be derived using congruence to λ-abstraction or extensionality. On the first case we know by induction hypothesis that $Q' \rhd Q^*$ and $(\lambda x.Q')N' \rhd Q^*[N^*/x]$. On the second case, we can take $P = \lambda x.Rx$, where, $R \rhd R'$. By induction, $(R'x) \rhd (Rx)^*$ and now we apply the substitution lemma to have $R'N' = (R'x)[N'/x] \rhd (Rx)^*[N^*/x]$.
- Congruence. Given $N \rhd N'$; by induction $N' \rhd N^*$, 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 $N \rhd N'$ could be derived by parallel application or parallel substitution. On the first case, given $P \rhd P'$, we know that $P' \rhd P^*$ by induction hypothesis and $\lambda x.P'x \rhd P^*$ by extensionality. On

the second case, $N = (\lambda y.Q)x$ and $N' = Q'[x/y]$, where $Q \rhd Q'$. Hence $P \rhd \lambda y.Q'$, and by induction hypothesis, $\lambda y.Q' \rhd P^*$.

- Parallel substitution, with $N \rhd N'$ and $Q \rhd Q'$; we know that $M^* = Q^*[N^*/x]$ and we can apply the substitution lemma (lemma 2) to get $M' \rhd M^*$.
- Extensionality. We know that $P \rhd P'$ and $x \notin \mathrm{FV}(P)$, so by induction hypothesis we know that $P' \rhd P^* = M^*$. □

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

Proof. Parallel reduction, $\rhd$, 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 $\rightarrow_{\beta\eta}$. □

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 $M \to_n N$ when N can be obtained by β-reducing the n-th leftmost β-reducible application of the expression. We call $\to_1$ the **leftmost one-step reduction** and we write it as $\to_l$; accordingly, $\twoheadrightarrow_l$ is its reflexive transitive closure.

Definition 13 (Standard sequence)**.** A sequence of β-reductions $M_0 \to_{n_1} M_1 \to_{n_2} M_2 \to_{n_3} \cdots \to_{n_k} M_k$ 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 $M \to_s \lambda x.B$.

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 $K \twoheadrightarrow_s M \to_\beta N$, 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 $M \to_\beta N$ can be derived.

1. If $K \twoheadrightarrow_s (\lambda x.M)N \to_\beta M[N/x]$, it has been already showed that $K \twoheadrightarrow_s M[N/x]$.
2. If $K \twoheadrightarrow_s MN \to_\beta M'N$ with $M \to_\beta M'$, 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 $K \twoheadrightarrow_s MN \to_\beta MN'$ is entirely analogous.
3. If $K \twoheadrightarrow_s \lambda x.M \to_\beta \lambda x.M'$, with $M \to_\beta M'$, 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;
- $\mathfrak{L}(I) = (\lambda x.x)$;
- $\mathfrak{L}(K) = (\lambda x.\lambda y.x)$;
- $\mathfrak{L}(S) = (\lambda x.\lambda y.\lambda z.xz(yz))$;
- $\mathfrak{L}(XY) = \mathfrak{L}(X)\mathfrak{L}(Y)$.

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 $x \notin \mathrm{FV}(U)$;
- $[x].UV = S([x].U)([x].V)$, 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 ”[BROKEN LINK: *Programming in the untyped \lambda-calculus]”.

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 $A \to B$ 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* $\to$. 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.

Definition 17 (Simple types)**.** The set of **simple types** is given by the following Backus-Naur form

$$\texttt{Type} ::= \iota \mid \texttt{Type} \to \texttt{Type},$$

where ι would be any *basic type*.

That is to say that, for every two types A, B, there exists a **function type** $A \to B$ between them.

2.2 TYPING RULES FOR SIMPLY TYPED λ-CALCULUS

We will now define the terms of simply typed λ-calculus using the same constructors we used on the untyped version. Those are the ***raw typed λ-terms***.

Definition 18 (Raw typed lambda terms)**.** The set of **typed lambda terms** is given by the following Backus-Naus form

$$\texttt{Term} ::= x \mid \texttt{Term Term} \mid \lambda x^{\texttt{Type}}.\texttt{Term}.$$

The main difference here with Definition 1 is that every bound variable has a type, and therefore, every λ-abstraction of the form $(\lambda x^A.M)$ can be applied only over terms type A; if M is of type B, this term will be of type $A \to B$.

However, the set of raw typed λ-terms contains some meaningless terms under this type interpretation, such as $(\lambda x^A.M)(\lambda x^A.M)$.[1] **Typing rules** will give them the desired expressive power; only a subset of these raw lambda terms will be typeable, and we will choose to work only with that subset. When a particular term M has type A, we write this relation as $M : A$, where the : symbol should be read as "is of type".

Definition 19 (Typing context)**.** A **typing context** is a sequence of type assumptions $x_1 : A_1, \dots, x_n : A_n$, where no variable x_i appears more than once. We will implicitely assume that the order in which these assumptions appear does not matter.

Every typing rule assumes a typing context, usually denoted by Γ. Concatenation of typing contexts is written as Γ, Γ'; and the fact that ψ follows from Γ is written as $\Gamma \vdash \psi$. Typing rules are written as rules of inference; the premises are listed above and the conclusion is written below the line.

[1] : In particular, we can not apply a function of type $A \to B$ to a term of type $A \to B$; it is expecting a term of type A.

1. The (var) rule simply makes explicit the type of a variable from the context. That is, a context that assumes that $x : A$ can be written as $\Gamma, x : A$; and we can trivially deduce from it that $x : A$.

$$\dfrac{}{\Gamma, x : A \vdash x : A}\;(var)$$

2. The (abs) rule declares that the type of a λ-abstraction is the type of functions from the variable type to the result type. If a term $M : B$ can be built from the assumption that $x : A$, then $\lambda x^A.M : A \to B$. It acts as a *constructor* of function terms.

$$\dfrac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\;(abs)$$

3. The (app) rule declares the type of a well-typed application. A term $f : A \to B$ applied to a term $a : A$ is a term $f\ a : B$. It acts as a *destructor* of function terms.

$$\dfrac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash fa : B}\;(app)$$

Definition 20. A term M is **typeable** in a giving context Γ if a typing judgement of the form $\Gamma \vdash M : T$ can be derived using only the previous typing rules.

From now on, we only consider typeable terms as the only terms of simply typed λ-calculus. As a consequence, the set of λ-terms of simply typed λ-calculus is only a subset of the terms of untyped λ-calculus.

Example 1 (Typeable and non-typeable terms). The term $\lambda f.\lambda x.f(fx)$ is typeable. If we abbreviate $\Gamma = f : A \to A,\ x : A$, the detailed typing derivation can be written as

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : A \to A}\;(var) \qquad \dfrac{\dfrac{}{\Gamma \vdash x : A}\;(var) \qquad \dfrac{}{\Gamma \vdash f : A \to A}\;(var)}{\Gamma \vdash f\ x : A}\;(app)}{f : A \to A, x : A \vdash f(fx) : A}\;(app)}{f : A \to A \vdash \lambda x.f(fx) : A \to A}\;(abs)}{\vdash \lambda f.\lambda x.f(fx) : (A \to A) \to A \to A}\;(abs)$$

The term $(\lambda x.x\ x)$, however, is not typeable. If x were of type ψ, it also should be of type $\psi \to \sigma$ for some σ in order for $x\ x$ to be well-typed; but $\psi \equiv \psi \to \sigma$ is not solvable, as it can be shown by structural induction on the term ψ.

It can be seen that the typing derivation of a term somehow encodes the complete λ-term. If we were to derive the term bottom-up, there would be only one possible choice at each step on which rule to use. In the following sections we will discuss a type inference algorithm that determines if a type is typeable and what its type should be, and we will make precise these intuitions.

2.3 CURRY-STYLE TYPES

Two different approaches to typing in λ-calculus are commonly used.

- **Church-style** typing, also known as *explicit typing*, originated from the work of Alonzo Church in [Chu40], where he described a simply-typed lambda calculus with two basic types. The term's type is defined as an intrinsic property of the term; and the same term has to be always interpreted with the same type.
- **Curry-style** typing, also known as *implicit typing*; which creates a formalism where every single term can be given an infinite number of types. This technique is called ***polymorphism*** when it is a formal part of the language; but here, it is only used to allow us to build intermediate terms without having to directly specify their type.

As an example, we can consider the identity term $I = \lambda x.x$. It would have to be defined for each possible type. That is, we should consider a family of different identity terms $I_A = \lambda x.x : A \to A$. Curry-style typing allows us to consider parametric types with type variables, and to type the identity as $I = \lambda x.x : \sigma \to \sigma$ where σ would a free type variable.

Definition 21 (Parametric types)**.** Given a infinite numerable set of *type variables*, we define **parametric types** or **type templates** inductively as

$$\mathtt{PType} ::= \iota \mid \mathtt{Tvar} \mid \mathtt{PType} \to \mathtt{PType},$$

where ι is a basic type, `Tvar` is a type variable and `PType` is a parametric type. That is, all basic types and type variables are atomic parametric types; and we also consider the arrow type between two parametric types.

The difference between the two typing styles is then not a mere notational convention, but a difference on the expressive power that we assign to each term. The interesting property of type variables is that they can act as placeholders for other type templates. This is formalized with the notion of type substitution.

Definition 22 (Type substitution)**.** A **substitution** ψ is any function from type variables to type templates. Any substitution ψ can be extended to a function between type templates called $\overline{\psi}$ and defined inductively by

- $\overline{\psi}\iota = \iota$, for any basic type ι;
- $\overline{\psi}\sigma = \psi\sigma$, for any type variable σ;
- $\overline{\psi}(A \to B) = \overline{\psi}A \to \overline{\psi}B$.

That is, the parametric type $\overline{\psi}A$ is the same as A but with every type variable replaced according to the substitution ψ.

We consider a type to be *more general* than other if the latter can be obtained by applying a substitution to the former. In this case, the latter is called an *instance* of the former. For example, $A \to B$ is more general than its instance $(C \to D) \to B$, where

A has been substituted by $C \to D$. An crucial property of simply typed λ-calculus is that every type has a most general type, called its *principal type*; this will be proved in Theorem 5.

Definition 23 (Principal type)**.** A closed λ-term M has a **principal type** π if $M : \pi$ and given any $M : \tau$, we can obtain τ as an instance of π, that is, $\overline{\sigma}\pi = \tau$.

2.4 UNIFICATION AND TYPE INFERENCE

The unification of two type templates is the construction of two substitutions making them equal as type templates; that is, the construction of a type that is a particular instance of both at the same time. We will not only aim for an unifier but for the most general one between them.

Definition 24 (Most general unifier)**.** A substitution ψ is called an **unifier** of two sequences of type templates $\{A_i\}_{i=1,\dots,n}$ and $\{B_i\}_{i=1,\dots,n}$ if $\overline{\psi}A_i = \overline{\psi}B_i$ for any i. We say that it is the **most general unifier** if given any other unifier ϕ exists a substitution φ such that $\phi = \overline{\varphi} \circ \psi$.

Lemma 4 (Unification)**.** *If an unifier of $\{A_i\}_{i=1,\dots,n}$ and $\{B_i\}_{i=1,\dots,n}$ exists, the most general unifier can be found using the following recursive definition of* $\texttt{unify}(A_1,\dots,A_n;B_1,\dots,B_n)$.

1. $\texttt{unify}(x;x) = \text{id}$ *and* $\texttt{unify}(\iota,\iota) = \text{id}$;
2. $\texttt{unify}(x;B) = (x \mapsto B)$, *the substitution that only changes x by B; if x does not occur in B. The algorithm **fails** if x occurs in B;*
3. $\texttt{unify}(A;x)$ *is defined symmetrically;*
4. $\texttt{unify}(A \to A';B \to B') = \texttt{unify}(A,A';B,B')$;
5. $\texttt{unify}(A,A_1,\dots;B,B_1,\dots) = \overline{\psi} \circ \rho$ *where* $\rho = \texttt{unify}(A_1,\dots;B_1,\dots)$ *and* $\psi = \texttt{unify}(\overline{\rho}A;\overline{\rho}B)$;
6. $\texttt{unify}$ *fails in any other case;*

where x is any type variable. The two sequences $\{A_i\},\{B_i\}$ of types have no unifier if and only if $\texttt{unify}(\{A_i\};\{B_i\})$ *fails.*

Proof. It is easy to notice by structural induction that, if $\texttt{unify}(A;B)$ exists, it is in fact an unifier.

If the unifier fails in clause 2, there is obviously no possible unifier: the number of constructors on the first type template will be always smaller than the second one. If the unifier fails in clause 6, the type templates are fundamentally different, they have different head constructors and this is invariant to substitutions. This proves that the failure of the algorithm implies the non existence of an unifier.

We now prove that, if A and B can be unified, $\texttt{unify}(A,B)$ is the most general unifier. For instance, in the clause 2, if we call $\psi = (x \mapsto B)$ and, if η were another unifier, then $\eta x = \overline{\eta} x = \overline{\eta} B = \overline{\eta}(\psi(x))$; hence $\overline{\eta} \circ \psi = \eta$ by definition of ψ. A similar argument can be applied to clauses 3 and 4. In the clause 5, we suppose the existence of some unifier ψ'. The recursive call gives us the most general unifier ρ of $A_1, \dots, A_n$ and $B_1, \dots, B_n$; and since it is more general than ψ', there exists an α such that $\overline{\alpha} \circ \rho = \psi'$. Now, $\overline{\alpha}(\overline{\rho} A) = \psi'(A) = \psi'(B) = \overline{\alpha}(\overline{\rho} B)$, hence α is a unifier of $\overline{\rho} A$ and $\overline{\rho} B$; we can take the most general unifier to be ψ, so $\overline{\beta} \circ \psi = \overline{\alpha}$; and finally, $\overline{\beta} \circ (\overline{\psi} \circ \rho) = \overline{\alpha} \circ \rho = \psi'$.

We also need to prove that the unification algorithm terminates. Firstly, we note that every substitution generated by the algorithm is either the identity or it removes at least one type variable. We can perform induction on the size of the argument on all clauses except for clause 5, where a substitution is applied and the number of type variables is reduced. Therefore, we need to apply induction on the number of type variables and only then apply induction on the size of the arguments. □

Using unification, we can define type inference.

Theorem 5 (Type inference). *The algorithm* `typeinfer`(M, B), *defined as follows, finds the most general substitution σ such that $x_1 : \sigma A_1, \dots, x_n : \sigma A_n \vdash M : \overline{\sigma} B$ is a valid typing judgment if it exists; and fails otherwise.*

1. `typeinfer`$(x_i : A_i, \Gamma \vdash x_i : B) =$ `unify`(A_i, B);
2. `typeinfer`$(\Gamma \vdash MN : B) = \overline{\varphi} \circ \psi$, *where* $\psi =$ `typeinfer`$(\Gamma \vdash M : x \to B)$ *and* $\varphi =$ `typeinfer`$(\overline{\psi}\Gamma \vdash N : \overline{\psi} x)$ *for a fresh type variable x;*
3. `typeinfer`$(\Gamma \vdash \lambda x.M : B) = \overline{\varphi} \circ \psi$ *where* $\psi =$ `unify`$(B; z \to z')$ *and* $\varphi =$ `typeinfer`$(\overline{\psi}\Gamma, x : \overline{\psi} z \vdash M : \overline{\psi} z')$ *for fresh type variables z, z'.*

Note that the existence of fresh type variables is always asserted by the set of type variables being infinite. The output of this algorithm is defined up to a permutation of type variables.

Proof. The algorithm terminates by induction on the size of M. It is easy to check by structural induction that the inferred type judgments are in fact valid. If the algorithm fails, by Lemma 4, it is also clear that the type inference is not possible.

On the first case, the type is obviously the most general substitution by virtue of the previous Lemma 4. On the second case, if α were another possible substitution, in particular, it should be less general than ψ, so $\alpha = \beta \circ \psi$. As β would be then a possible substitution making $\overline{\psi}\Gamma \vdash N : \overline{\psi} x$ valid, it should be less general than φ, so $\alpha = \overline{\beta} \circ \psi = \overline{\gamma} \circ \overline{\varphi} \circ \beta$. On the third case, if α were another possible substitution, it should unify B to a function type, so $\alpha = \overline{\beta} \circ \psi$. Then β should make the type inference $\overline{\psi}\Gamma, x : \overline{\psi} z \vdash M : \overline{\psi} z'$ possible, so $\beta = \overline{\gamma} \circ \varphi$. We have proved that the inferred type is in general the most general one. □

Corollary 2 (Principal type property). *Every typeable pure λ-term has a principal type.*

Proof. Given a typeable term M, we can compute `typeinfer`$(x_1 : A_1, \dots, x_n : A_n \vdash M : B)$, where $x_1, \dots, x_n$ are the free variables on M and $A_1, \dots, A_n, B$ are fresh type variables. By virtue of Theorem 5, the result is the most general type of M if we assume the variables to have the given types. □

2.5 SUBJECT REDUCTION AND NORMALIZATION

A crucial property is that type inference and β-reductions do not interfere with each other. A term can be β-reduced without changing its type.

Theorem 6 (Subject reduction). *The type is preserved on β-reductions; that is, if $\Gamma \vdash M : A$ and and $M \twoheadrightarrow_\beta M'$, then $\Gamma \vdash M' : A$.*

Proof. If M' has been derived by β-reduction, $M = (\lambda x.P)$ and $M' = P[Q/x]$. $\Gamma \vdash M : A$ implies $\Gamma, x : B \vdash P : A$ and $\Gamma \vdash Q : B$. Again by structural induction on P (where the only crucial case uses that x and Q have the same type) we can prove that substitutions do not alter the type and thus, $\Gamma, Q : B \vdash P[Q/x] : A$. □

We have seen previously that the term $\Omega = (\lambda x.xx)(\lambda x.xx)$ is not weakly normalizing; but it is also non-typeable. In this section we will prove that, in fact, every typeable term is strongly normalizing. We start proving some lemmas about the notion of *reducibility*, which will lead us to the Strong Normalization Theorem. This proof will follow [GTL89].

The notion of ***reducibility*** is an abstract concept originally defined by Tait in [Tai67] which we will use to ease this proof. It should not be confused with the notion of β-reduction.

Definition 25 (Reducibility)**.** We inductively define the set of **reducible** terms of type T for basic and arrow types.

- If $t : T$ where T a basic type, $t \in \mathtt{RED}_T$ if t is strongly normalizable.
- If $t : U \to V$, an arrow type, $t \in \mathtt{RED}_{U \to V}$ if $t\ u \in \mathtt{RED}_V$ for all $u \in \mathtt{RED}_U$.

Properties of reducibility will be used directly in the Strong Normalization Theorem. We prove three of them at the same time in order to use mutual induction.

Proposition 1 (Properties of reducibility)**.** *The following three properties hold;*

1. *if $t \in \mathtt{RED}_T$, then t is strongly normalizable;*
2. *if $t \in \mathtt{RED}_T$ and $t \to_\beta t'$, $t' \in \mathtt{RED}_T$; and*
3. *if t is not a λ-abstraction and $t' \in \mathtt{RED}_T$ for every $t \to_\beta t'$, then $t \in \mathtt{RED}_T$.*

Proof. For basic types,

1. holds trivially;
2. holds by the definition of strong normalization;

3. if any one-step β-reduction leads to a strongly normalizing term, the term itself must be strongly normalizing.

For arrow types,

1. if $x : U$ is a variable, we can inductively apply (3) to get $x \in \mathtt{RED}_U$; then, $t\ x \in \mathtt{RED}_V$ is strongly normalizing and t in particular must be strongly normalizing;
2. if $t \to_\beta t'$ then for every $u \in \mathtt{RED}_U$, $t\ u \in \mathtt{RED}_V$ and $t\ u \to_\beta t'\ u$. By induction, $t'\ u \in \mathtt{RED}_V$;
3. if $u \in \mathtt{RED}_U$, it is strongly normalizable. As t is not a λ-abstraction, he term $t\ u$ can only be reduced to $t'\ u$ or $t\ u'$. If $t \to_\beta t'$; by induction, $t'\ u \in \mathtt{RED}_V$. If $u \to_\beta u'$, we could proceed by induction over the length of the longest chain of β-reductions starting from u and assume that $t\ u'$ is irreducible. In every case, we have proved that $t\ u$ only reduces to already reducible terms; thus, $t\ u \in \mathtt{RED}_U$.

□

Lemma 5 (Abstraction lemma). *If $v[u/x] \in \mathtt{RED}_V$ for all $u \in \mathtt{RED}_U$, then $\lambda x.v \in \mathtt{RED}_{U\to V}$.*

Proof. We apply induction over the sum of the lengths of the longest β-reduction sequences from $v[x/x]$ and u. The term $(\lambda x.v)u$ can be β-reduced to

- $v[u/x] \in \mathtt{RED}_U$; in the base case of induction, this is the only choice;
- $(\lambda x.v')u$ where $v \to_\beta v'$, and, by induction, $(\lambda x.v')u \in \mathtt{RED}_V$;
- $(\lambda x.v)u'$ where $u \to_\beta u'$, and, again by induction, $(\lambda x.v)u' \in \mathtt{RED}_V$.

Thus, by Proposition 1, $(\lambda x.v) \in \mathtt{RED}_{U\to V}$. □

A final lemma is needed before the proof of the Strong Normalization Theorem. It is a generalization of the main theorem, useful because of the stronger induction hypothesis it provides.

Lemma 6 (Strong Normalization lemma). *Given an arbitrary $t : T$ with free variables $x_1 : U_1, \dots, x_n : U_n$, and reducible terms $u_1 \in \mathtt{RED}_{U_1}, \dots, u_n \in \mathtt{RED}_{U_2}$, we know that*

$$t[u_1/x_1][u_2/x_2]\dots[u_n/x_n] \in \mathtt{RED}_T.$$

Proof. We call $\tilde{t} = t[u_1/x_1][u_2/x_2]\dots[u_n/x_n]$ and apply structural induction over t,

- if $t = x_i$, then we simply use that $u_i \in \mathtt{RED}_{U_i}$,
- if $t = v\ w$, then we apply induction hypothesis to get $\tilde{v} \in \mathtt{RED}_{R\to T}, \tilde{w} \in \mathtt{RED}_R$ for some type R. Then, by definition, $\tilde{t} = \tilde{v}\ \tilde{w} \in \mathtt{RED}_T$,
- if $t = \lambda y.v : R \to S$, then by induction $\tilde{v}[r/y] \in \mathtt{RED}_S$ for every $r : R$. We can then apply Lemma 5 to get that $\tilde{t} = \lambda y.\tilde{v} \in \mathtt{RED}_{R\to S}$. □

Theorem 7 (Strong Normalization Theorem). *In simply typed λ-calculus, all terms are strongly normalizing.*

Proof. It is the particular case of Lemma 6 where we take $u_i = x_i$. □

Every term normalizes in simply typed λ-calculus and every computation ends. We know, however, that the Halting Problem is unsolvable, so simply typed λ-calculus must be not Turing complete.

3

THE CURRY-HOWARD CORRESPONDENCE

3.1 EXTENDING THE SIMPLY TYPED λ-CALCULUS

We will add now special syntax for some terms and types, such as pairs, unions and unit types. This syntax will make our λ-calculus more expressive, but the unification and type inference algorithms will continue to work. The previous proofs and algorithms can be extended to cover all the new cases.

Definition 26 (Simple types II)**.** The new set of **simple types** is given by the following BNF

$$\texttt{Type} ::= \iota \mid \texttt{Type} \to \texttt{Type} \mid \texttt{Type} \times \texttt{Type} \mid \texttt{Type} + \texttt{Type} \mid 1 \mid 0,$$

where ι would be any *basic type*.

That is to say that, for any given types A, B, there exists a product type $A \times B$, consisting of the pairs of elements where the first one is of type A and the second one of type B; there exists the union type $A + B$, consisting of a disjoint union of tagged terms from A or B; an unit type 1 with only an element, and an empty or void type 0 without inhabitants. The raw typed λ-terms are extended to use these new types.

Definition 27 (Raw typed lambda terms II)**.** The new set of raw **typed lambda terms** is given by the BNF

$$\begin{aligned}\texttt{Term} ::= {} & x \mid \texttt{Term}\texttt{Term} \mid \lambda x.\texttt{Term} \mid \\ & \langle \texttt{Term}, \texttt{Term} \rangle \mid \pi_1 \texttt{Term} \mid \pi_2 \texttt{Term} \mid \\ & \text{inl}\ \texttt{Term} \mid \text{inr}\ \texttt{Term} \mid \text{case}\ \texttt{Term}\ \text{of}\ \texttt{Term}; \texttt{Term} \mid \\ & \text{abort}\ \texttt{Term} \mid *\end{aligned}$$

The use of these new terms is formalized by the following extended set of typing rules.

1. The (var) rule simply makes explicit the type of a variable from the context.

$$(var)\ \frac{}{\Gamma, x : A \vdash x : A}$$

2. The (abs) gives the type of a λ-abstraction as the type of functions from the variable type to the result type. It acts as a constructor of function terms.

$$(abs)\,\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}$$

3. The (app) rule gives the type of a well-typed application of a lambda term. A term $f : A \to B$ applied to a term $a : A$ is a term of type B. It acts as a destructor of function terms.

$$(app)\,\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash fa : B}$$

4. The $(pair)$ rule gives the type of a pair of elements. It acts as a constructor of pair terms.

$$(pair)\,\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \langle a, b \rangle : A \times B}$$

5. The (π_1) rule extracts the first element from a pair. It acts as a destructor of pair terms.

$$(\pi_1)\,\frac{\Gamma \vdash m : A \times B}{\Gamma \vdash \pi_1\ m : A}$$

6. The (π_1) rule extracts the second element from a pair. It acts as a destructor of pair terms.

$$(\pi_2)\,\frac{\Gamma \vdash m : A \times B}{\Gamma \vdash \pi_2\ m : B}$$

7. The (inl) rule creates a union type from the left side type of the sum. It acts as a constructor of union terms.

$$(inl)\,\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathrm{inl}\ a : A + B}$$

8. The (inr) rule creates a union type from the right side type of the sum. It acts as a constructor of union terms.

$$(inr)\,\frac{\Gamma \vdash b : B}{\Gamma \vdash \mathrm{inr}\ b : A + B}$$

9. The $(case)$ rule extracts a term from an union and applies the appropiate deduction on any of the two cases

$$(case)\,\frac{\Gamma \vdash m : A + B \qquad \Gamma, a : A \vdash n : C \qquad \Gamma, b : B \vdash p : C}{\Gamma \vdash (\mathrm{case}\ m\ \mathrm{of}\ [a].n;\ [b].p) : C}$$

   Note that we write $[a].n$ and $[b].p$ to indicate that n and p depend on a and b respectively.

10. The $(*)$ rule simply creates the only element of 1. It is a constructor of the unit type.

$$(*)\,\frac{}{\Gamma \vdash * : 1}$$

11. The $(abort)$ rule extracts a term of any type from the void type.

$$(abort)\ \frac{\Gamma \vdash M : 0}{\Gamma \vdash \text{abort}_A\ M : A}$$

The abort function must be understood as the unique function going from the empty set to any given set.

The β-reduction of terms is defined the same way as for the untyped λ-calculus; except for the inclusion of β-rules governing the new terms, each for every new destruction rule.

1. Function application, $(\lambda x.M)N \to_\beta M[N/x]$.
2. First projection, $\pi_1 \langle M, N \rangle \to_\beta M$.
3. Second projection, $\pi_2 \langle M, N \rangle \to_\beta N$.
4. Case rule, $(\text{case } m \text{ of } [a].N;\ [b].P) \to_\beta Na$ if m is of the form $m = \text{inl } a$; and $(\text{case } m \text{ of } [a].N;\ [b].P) \to_\beta Pb$ if m is of the form $m = \text{inr } b$.

On the other side, new η-rules are defined, each for every new construction rule.

1. Function extensionality, $\lambda x.Mx \to_\eta M$.
2. Definition of product, $\langle \pi_1 M, \pi_2 M \rangle \to_\eta M$.
3. Uniqueness of unit, $M \to_\eta *$.
4. Case rule, $(\text{case } m \text{ of } [a].P[\text{inl } a/c];\ [b].P[\text{inr } b/c]) \to_\eta P[m/c]$.

3.2 natural deduction

The natural deduction is a logical system due to Gentzen. We introduce it here following [Sel13] and [Wad15]. Its relationship with the simply-typed lambda calculus will be made explicit in the next section.

We will use the logical binary connectives $\to, \wedge, \vee$, and two given propositions, $\top, \bot$ representing the trivially true and false propostiions, respectively. The rules defining natural deduction come in pairs; there are introductors and eliminators for every connective. Every introductor uses a set of assumptions to generate a formula and every eliminator gives a way to extract precisely that set of assumptions.

1. Every axiom on the context can be used.

$$\frac{}{\Gamma, A \vdash A}\ (\text{Ax})$$

2. Introduction and elimination of the $\to$ connective. Note that the elimination rule corresponds to *modus ponens* and the introduction rule corresponds to the *deduction theorem*.

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ (I_\to) \qquad \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ (E_\to)$$

3. Introduction and elimination of the $\wedge$ connective. Note that the introduction in this case takes two assumptions, and there are two different elimination rules.

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\,(I_\wedge) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\,(E^1_\wedge) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\,(E^2_\wedge)$$

4. Introduction and elimination of the $\vee$ connective. Here, we need two introduction rules to match the two assumptions we use on the eliminator.

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\,(I^1_\vee) \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\,(I^2_\vee) \qquad \frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}\,(E_\vee)$$

5. Introduction for $\top$. It needs no assumptions and, consequently, there is no elimination rule for it.

$$\frac{}{\Gamma \vdash \top}\,(I_\top)$$

6. Elimination for $\bot$. It can be eliminated in all generality, and, consequently, there are no introduction rules for it. This elimination rule represents the *"ex falsum quodlibet"* principle that says that falsity implies anything.

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash C}\,(E_\bot)$$

Proofs on natural deduction are written as deduction trees, and they can be simplified according to some simplification rules, which can be applied anywhere on the deduction tree. On these rules, a chain of dots represents any given part of the deduction tree.

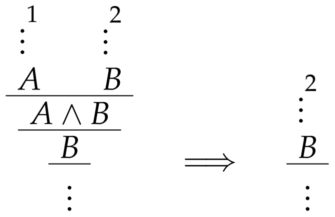
1. An implication and its antecedent can be simplified using the antecedent directly on the implication.

$$\begin{array}{c}\dfrac{\dfrac{\begin{array}{c}[A]\\ \overset{1}{\vdots}\\ B\end{array}}{A \to B} \qquad \begin{array}{c}\overset{2}{\vdots}\\ A\end{array}}{\underline{\;B\;}}\\ \vdots\end{array} \quad \Longrightarrow \quad \begin{array}{c}\overset{2}{\vdots}\\ A\\ \overset{1}{\vdots}\\ \underline{\;B\;}\\ \vdots\end{array}$$

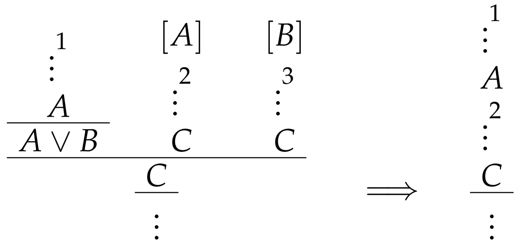
2. The introduction of an unused conjunction can be simplified as

$$\begin{array}{c}\dfrac{\dfrac{\begin{array}{cc}\overset{1}{\vdots} & \overset{2}{\vdots}\\ A & B\end{array}}{A \wedge B}}{\underline{\;A\;}}\\ \vdots\end{array} \quad \Longrightarrow \quad \begin{array}{c}\overset{1}{\vdots}\\ \underline{\;A\;}\\ \vdots\end{array}$$

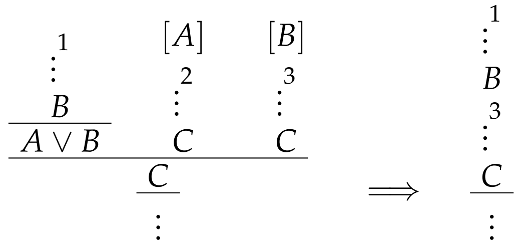
and, similarly, on the other side as



3. The introduction of a disjunction followed by its elimination can be also simplified



and a similar pattern is used on the other side of the disjunction



3.3 PROPOSITIONS AS TYPES

In 1934, Curry observed in [Cur34] that the type of a function $(A \to B)$ could be read as an implication and that the existence of a function of that type was equivalent to the provability of the proposition. Previously, the **Brouwer-Heyting-Kolmogorov interpretation** of intuitionistic logic had given a definition of what it meant to be a proof of an intuinistic formula, where a proof of the implication $(A \to B)$ was a function converting a proof of A into a proof of B. It was not until 1969 that Howard pointed a deep correspondence between the simply-typed λ-calculus and the natural deduction at three levels

1. propositions are types.
2. proofs are programs.

3. simplification of proofs is the evaluation of programs.

In the case of simply typed λ-calculus and natural deduction, the correspondence starts when we describe the following one-to-one relation between types and propositions.

Types	Propositions
Unit type (1)	Truth ($\top$)
Product type ($\times$)	Conjunction ($\wedge$)
Union type ($+$)	Disjunction ($\vee$)
Function type ($\to$)	Implication ($\to$)
Empty type (0)	False ($\perp$)

Where, in particular, the negation of a proposition $\neg A$ is interpreted as the fact that that proposition implies falsehood, $A \to \perp$; and its corresponding type is a function from the type A to the empty type, $A \to 0$.

Now it is easy to notice that every deduction rule for a proposition has a correspondence with a typing rule. The only distinction between them is the appearance of λ-terms on the first set of rules. As every typing rule results on the construction of a particular kind of λ-term, they can be interpreted as encodings of proof in the form of derivation trees. That is, terms are proofs of the propositions represented by their types.

Example 2 (Curry-Howard correspondence example). In particular, the typing derivation of the term

$$\lambda a.\lambda b.\langle a, b\rangle$$

can be seen as a deduction tree proving $A \to B \to A \wedge B$; as the following diagram shows, in which terms are colored in red and types are colored in *blue*.

$$\dfrac{\dfrac{\dfrac{\texttt{a} :: A \qquad \texttt{b} :: B}{\texttt{<a,b>} :: A \times B}\,(pair)}{\lambda\texttt{b.<a,b>} :: B \to A \times B}\,(abs)}{\lambda\texttt{a.}\lambda\texttt{b.<a,b>} :: A \to B \to A \times B}\,(abs)$$

Furthermore, under this interpretation, ***simplification rules are precisely β-reduction rules***. This makes execution of λ-calculus programs correspond to proof simplification on natural deduction. The Curry-Howard correspondence is then not only a simple bijection between types and propositions, but a deeper isomorphism regarding the way they are constructed, used in derivations, and simplified.

Example 3 (Curry-Howard simplification example). As an example of this duality, we will write a proof/term of the proposition/type `A → B + A` and we are going to simplify/compute it using proof simplification rules/β-rules. Similar examples can be found in [Wad15].

We start with the following derivation tree; in which terms are colored in red and types are colored in *blue*

$$\dfrac{\dfrac{\dfrac{\texttt{b} :: [A+B] \qquad \dfrac{\texttt{c} :: A}{\texttt{inr c} :: B+A}\,(inr) \qquad \dfrac{\texttt{c} :: B}{\texttt{inl c} :: B+A}\,(inl)}{\texttt{case b of [c].inr c; [c].inl c} :: B+A}\,(case)}{\texttt{λb.case b of [c].inr c; [c].inl c} :: A+B \to B+A}\,(abs) \qquad \dfrac{\texttt{a} :: A}{\texttt{inl a} :: A+B}\,(inl)}{\dfrac{\texttt{(λb.case b of [c].inr c; [c].inl c)(inl a)} :: B+A}{\texttt{λa.(λb.case b of [c].inr c; [c].inl c) (inl a)} :: A \to B+A}\,(abs)}\,(app)$$

which is encoded by the term `λa.(λc.case c of [a].inr a; [b].inl b) (λz.inl z)`. We apply the simplification rule/β-rule of the implication/function application to get

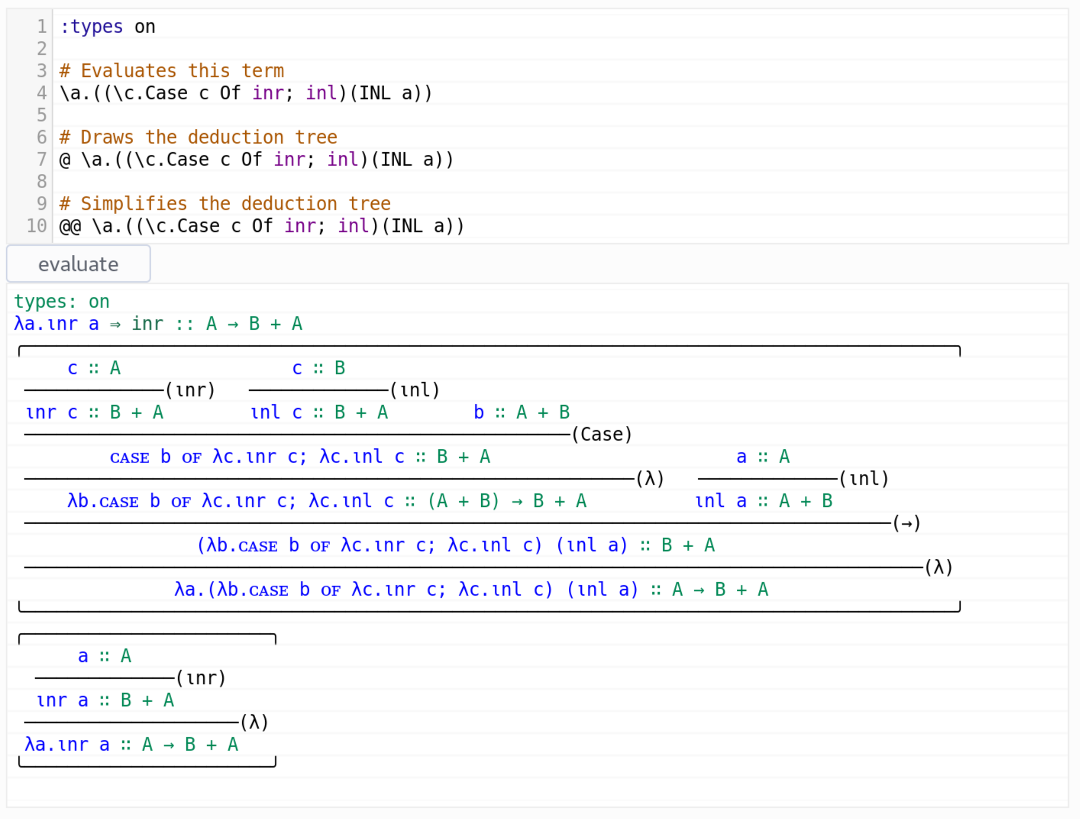
$$\dfrac{\dfrac{\dfrac{\texttt{z} :: A}{\texttt{inl z} :: A+B}\,(inl) \qquad \dfrac{\texttt{a} :: A}{\texttt{inr a} :: B+A}\,(inr) \qquad \dfrac{\texttt{b} :: B}{\texttt{inl b} :: B+A}\,(inl)}{\texttt{case (inl z) of [a].inr a; [b].inl b} :: B+A}\,(case)}{\texttt{λ z.case (inl z) of [a].inr a; [b].inl b} :: A \to B+A}\,(abs)$$

which is encoded by the term `λa.case (inl a) of (inr) (inl)`. We finally apply the `case` simplification/reduction rule to get

$$\dfrac{\dfrac{\texttt{a} :: A}{\texttt{inr a} :: B+A}\,(inr)}{\texttt{λ a.inr a} :: A \to B+A}\,(abs)$$

which is encoded by `λa.(inr a)`.

On the chapter on Mikrokosmos, we develop a λ-calculus interpreter which is able to check and simplify proofs in intuitionistic logic. This example could be checked and simplified by this interpreter as it is shown in image 1.



```
:types on

# Evaluates this term
\a.((\c.Case c Of inr; inl)(INL a))

# Draws the deduction tree
@ \a.((\c.Case c Of inr; inl)(INL a))

# Simplifies the deduction tree
@@ \a.((\c.Case c Of inr; inl)(INL a))
```

```
types: on
λa.ιnr a ⇒ inr :: A → B + A
┌───────────────────────────────────────────────────────────────────────────────────────┐
    c ∷ A                c ∷ B
─────────────(ιnr)   ─────────────(ιnl)
ιnr c ∷ B + A        ιnl c ∷ B + A        b ∷ A + B
────────────────────────────────────────────────────(Case)
        CASE b OF λc.ιnr c; λc.ιnl c ∷ B + A                       a ∷ A
──────────────────────────────────────────────────────────(λ)   ─────────────(ιnl)
    λb.CASE b OF λc.ιnr c; λc.ιnl c ∷ (A + B) → B + A          ιnl a ∷ A + B
──────────────────────────────────────────────────────────────────────────────────(→)
                (λb.CASE b OF λc.ιnr c; λc.ιnl c) (ιnl a) ∷ B + A
─────────────────────────────────────────────────────────────────────────────────────(λ)
              λa.(λb.CASE b OF λc.ιnr c; λc.ιnl c) (ιnl a) ∷ A → B + A
└───────────────────────────────────────────────────────────────────────────────────────┘

┌────────────────────────┐
     a ∷ A
 ─────────────(ιnr)
 ιnr a ∷ B + A
────────────────────(λ)
λa.ιnr a ∷ A → B + A
└────────────────────────┘
```

Figure 1: Curry-Howard example in Mikrokosmos.

4

OTHER TYPE SYSTEMS

4.1 λ-cube

The λ**-cube** is a taxonomy for Church-style type systems given by Barendregt in [Bar92]. It describes eight type systems based on the λ-calculus along three axes, representing three properties of the systems. These properties are

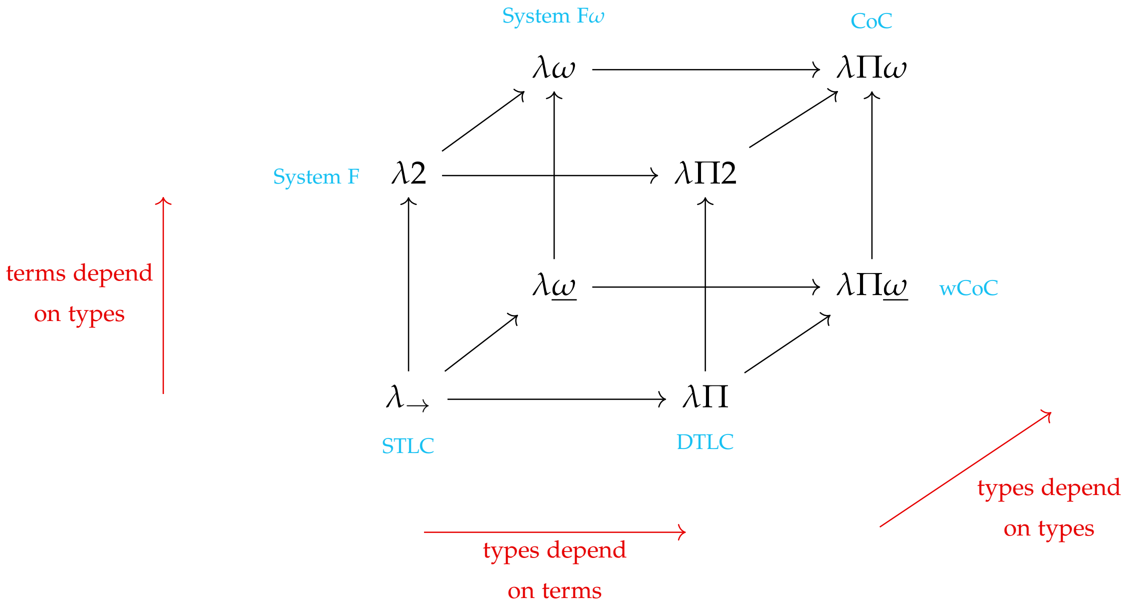
1. **parametric polymorphism**, terms that depend on types. This is achieved via universal quantification over types. It allows type variables and binders for them. An example is the following parametric identity function

   $$\mathrm{id} \equiv \Lambda\tau.\lambda x.x : \forall\tau.\tau \to \tau,$$

   that can be applied to any particular type σ to obtain the specific identity function for that type as

   $$\mathrm{id}_\sigma \equiv \lambda x.x : \sigma \to \sigma.$$

   System F is the simplest type system on the cube implementing polymorphism.
2. **type operators**, types that depend on types.
3. **dependent types**, types that depend on terms.



The following type systems

- **Simply typed λ-calculus** ($\lambda_\rightarrow$);
- **System F** ($\lambda 2$);
- typed λ-calculus with **dependent types** ($\lambda\Pi$);
- typed λ-calculus with **type operators** ($\lambda\underline{\omega}$);
- **System F-omega** ($\lambda\omega$);

The λ-cube is generalized by the theory of pure type systems.

All systems on the λ-cube are strongly normalizing.

A different approach to higher-order type systems will be presented in the chapter on Type Theory.

Part II

MIKROKOSMOS

We have developed **Mikrokosmos**, an untyped and simply typed λ-calculus interpreter written in the purely functional programming language Haskell [HHJW07]. It aims to provide students with a tool to learn and understand λ-calculus and the relation between logic and types.

5

IMPLEMENTATION OF λ-EXPRESSIONS

5.1 THE HASKELL PROGRAMMING LANGUAGE

Haskell is the purely functional programming language of our choice to implement Mikrokosmos, our λ-calculus interpreter. Its own design is heavily influenced by the λ-calculus and is a general-purpose language with a rich ecosystem and plenty of consolidated libraries[1] in areas such as parsing, testing or system interaction; matching the requisites of our project. In the following sections, we describe this ecosystem in more detail.

In the 1980s, many lazy programming languages were independently being written by researchers such as *Miranda*, *Lazy ML*, *Orwell*, *Clean* or *Daisy*. All of them were similar in expressive power, but their differences were holding back the efforts to communicate ideas on functional programming. A comitee was created in 1987 with the mission of designing a common lazy functional language. Several versions of the language were developed, and the first standarized reference of the language was published in the **Haskell 98 Report**, whose revised version can be read in [P+03]. Its more popular implementation is the **Glasgow Haskell Compiler (GHC)**; an open source compiler written in Haskell and C. The complete history of Haskell and its design decisions is detailed on [HHJW07]. Haskell is

1. **strongly and statically typed**, meaning that it only compiles well-typed programs and it does not allow implicit type casting; the compiler will generate an error if a term is non-typeable;
2. **lazy**, with *non-strict semantics*, meaning that it will not evaluate a term or the argument of a function until it is needed; in [Hug89], John Hughes, codesigner of the language, argues for the benefits of a lazy functional language, which could solve the traditional efficiency problems on functional programming;
3. **purely functional**; as the evaluation order is demand-driven and not explicitly known, it is not possible in practice to perform ordered input/output actions or

[1] : In the central package archive of the Haskell community, Hackage, a categorized list of libraries can be found: `https://hackage.haskell.org/packages/`

any other side-effects by relying on the evaluation order; this helps modularity of the code, testing, and verfication;

4. **referentially transparent**; as a consequence of its purity, every term on the code could be replaced by its definition without changing the global meaning of the program; this allows equational reasoning with rules that are directly derived from λ-calculus;
5. based on **System Fω** with some restrictions; crucially, it implements **System F** adding quantification over type operators even if it does not allow abstraction on type operators; the GHC Haskell compiler, however, allows the user to activate extensions that implement dependent types.

Example 4 (A first example in Haskell)*.* This example shows the basic syntax and how its type system and its implicit laziness can be used.

```haskell
-- The type of the term can be declared.
id :: a -> a  -- Polymorphic type variables are allowed,
id x = x      -- and the function is defined equationally.
-- This definition performs short circuit evaluation thanks
-- to laziness. The unused argument can be omitted.
(&&) :: Bool -> Bool -> Bool
True  && x = x                -- (true and x) is always x
False && _ = False            -- (false and y) is always false
-- Laziness also allows infinite data structures.
nats :: [Integer]         -- List of all natural numbers,
nats = 1 : map (+1) nats  -- defined recursively.
```

Where most imperative languages use semicolons to separate sequential commands, Haskell has no notion of sequencing, and programs are written in a purely declarative way. A Haskell program essentially consist on a series of definitions (of both types and terms) and type declarations. The following example shows the definition of a binary tree and its preorder as

```haskell
-- A tree is either empty or a node with two subtrees.
data Tree a = Empty | Node a (Tree a) (Tree a)
-- The preorder function takes a tree and returns a list
preorder :: Tree a -> [a]
preorder Empty            = []
preorder (Node x lft rgt) = preorder lft ++ [x] ++ preorder rgt
```

We can see on the previous example that function definitions allow *pattern matching*, that is, data constructors can be used in definitions to decompose values of the type. This increases readability when working with algebraic data types.

While infix operators are allowed, function application is left-associative in general. Definitions using partial application are allowed, meaning that functions on multiple arguments can use currying and can be passed only one of its arguments to define a new function. For example, a function that squares every number on a list could be written in two ways as

```
squareList :: [Int] -> [Int]
squareList list = map square list
squareList' :: [Int] -> [Int]
squareList' = map square
```

where the second one, because of its simplicity, is usually preferred. A characteristic piece of Haskell are **type classes**, which allow defining common interfaces for different types. In the following example, we define `Monad` as the type class of types with suitably typed `return` and `bind` operators.

```
class Monad m where
  return :: a   -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

And lists, for example, are monads in this sense.

```
instance Monad [] where
  return x = [x]               -- returns a one-element list
  xs >>= f = concat (map f xs) -- map and concatenation
```

Haskell uses monads in varied forms. They are used in I/O, error propagation and stateful computations. Another characteristical syntax bit of Haskell is the `do` notation, which provides a nicer, cleaner way to work with types that happen to be monads. The following example uses the list monad to compute the list of Pythagorean triples.

```
pythagorean = do
  a <- [1..]                -- let a be any natural
  b <- [1..a]               -- let b be a natural between 1 and a
  c <- [1..b]               -- let c be a natural between 1 and b
  guard (a^2 == b^2 + c^2)  -- filter the list
  return (a,b,c)            -- return matching tuples
```

Note that this list is infinite. As the language is lazy, this does not represent a problem: the list will be evaluated only on demand.

Another common example of an instance of the `Monad` typeclass is the *Maybe monad* used to deal with error propagation. A `Maybe a` type can consist of a term of type `a`, written as `Just a`; or as a `Nothing` constant, signalling an error. The monad is then defined as

```
instance Monad Maybe where
  return x = Just x
  xs >>= f = case xs of Nothing -> Nothing | Just a -> Just (f a)
```

and can be used as in the following example to use *exception-like* error treatment in a pure declarative language

```
roots :: (Float,Float,Float) -> Maybe Int
roots (a,b,c) = do
  -- Some errors can occur during this computation
  discriminant <- sqroot (b*b - 4*c*a)          -- roots of negative numbers?
  root1 <- safeDiv ((-b) + discriminant) (2*a) -- division by zero?
  root2 <- safeDiv ((-b) - discriminant) (2*a)
  -- The monad ensures that we return a number only if no error has been raised
  return (root1,root2)
```

For a more detailed treatment of monads, and their relation to categorical monads, see the chapter on Category Theory and the chapter on Type Theory, where we will program with monads in Agda.

5.2 DE BRUIJN INDEXES

Nicolaas Govert **De Bruijn** proposed in [dB72] a way of defining λ-terms modulo α-conversion based on indices. The main goal of De Bruijn indices is to remove all variables from binders and replace every variable on the body of an expression with a number, called *index*, representing the number of λ-abstractions in scope between the ocurrence and its binder.

Consider the following example: the λ-term

$$\lambda x.(\lambda y.\ y(\lambda z.\ yz))(\lambda t.\lambda z.\ tx)$$

can be written with de Bruijn indices as

$$\lambda\ (\lambda(1\lambda(21))\ \lambda\lambda(23)\ ).$$

De Bruijn also proposed a notation for the λ-calculus changing the order of binders and λ-applications. A review on the syntax of this notation, its advantages and De Bruijn indexes, can be found in [Kam01]. In this section, we are going to describe De Bruijn indexes while preserving the usual notation of λ-terms; that is, the *De Bruijn indexes* and the *De Bruijn notation* are different concepts and we are going to use only the former.

Definition 28 (De Bruijn indexed terms)**.** We define recursively the set of λ-terms using de Bruijn notation following this BNF

$$\mathtt{Exp} ::= \underbrace{\mathbb{N}}_{\textit{variable}} \mid \underbrace{(\lambda\ \mathtt{Exp})}_{\textit{abstraction}} \mid \underbrace{(\mathtt{Exp}\ \mathtt{Exp})}_{\textit{application}}$$

$$\mathbb{N} ::= 0 \mid 1 \mid 2 \mid \dots$$

Our internal definition closely matches the formal one. The names of the constructors here are `Var`, `Lambda` and `App`:

```haskell
-- | A lambda expression using DeBruijn indexes.
data Exp = Var Integer -- ^ integer indexing the variable.
         | Lambda Exp  -- ^ lambda abstraction
         | App Exp Exp -- ^ function application
         deriving (Eq, Ord)
```

This notation avoids the need for the Barendregt's variable convention and the α-reductions. It will be useful to implement λ-calculus without having to worry about the specific names of variables.

5.3 substitution

We define the substitution operation needed for the β-reduction on de Bruijn indices. In order to define the substitution of the n-th variable by a λ-term P on a given term, we must

- find all the ocurrences of the variable. At each level of scope we are looking for the successor of the number we were looking for before;
- decrease the higher variables to reflect the disappearance of a lambda;
- replace the ocurrences of the variables by the new term, taking into account that free variables must be increased to avoid them getting captured by the outermost lambda terms.

In our code, we apply `subs` to any expression. When it is applied to a λ-abstraction, the index and the free variables of the replaced term are increased with `incrementFreeVars`; whenever it is applied to a variable, the previous cases are taken into consideration.

```haskell
-- | Substitutes an index for a lambda expression
subs :: Integer -> Exp -> Exp -> Exp
subs n p (Lambda e) = Lambda (subs (n+1) (incrementFreeVars 0 p) e)
subs n p (App f g)  = App (subs n p f) (subs n p g)
subs n p (Var m)
  | n == m    = p          -- The lambda is replaced directly
  | n <  m    = Var (m-1) -- A more exterior lambda decreases a number
  | otherwise = Var m     -- An unrelated variable remains untouched
```

Then β-reduction can be defined using this subs function.

```
betared :: Exp -> Exp
betared (App (Lambda e) x) = substitute 1 x e
betared e = e
```

5.4 DE BRUIJN-TERMS AND λ-TERMS

The internal language of the interpreter uses de Bruijn expressions, while the user interacts with it using lambda expressions with alphanumeric variables. Our definition of a λ-expression with variables will be used in parsing and output formatting.

```
data NamedLambda = LambdaVariable String
                 | LambdaAbstraction String NamedLambda
                 | LambdaApplication NamedLambda NamedLambda
```

The translation from a natural λ-expression to de Bruijn notation is done using a dictionary which keeps track of the bounded variables

```
tobruijn :: Map.Map String Integer -- ^ names of the variables used
         -> Context                -- ^ names already binded on the scope
         -> NamedLambda            -- ^ initial expression
         -> Exp
-- Every lambda abstraction is inserted in the variable dictionary,
-- and every number in the dictionary increases to reflect we are entering
-- a deeper context.
tobruijn d context (LambdaAbstraction c e) =
     Lambda $ tobruijn newdict context e
        where newdict = Map.insert c 1 (Map.map succ d)

-- Translation distributes over applications.
tobruijn d context (LambdaApplication f g) =
     App (tobruijn d context f) (tobruijn d context g)

-- We look for every variable on the local dictionary and the current scope.
tobruijn d context (LambdaVariable c) =
  case Map.lookup c d of
    Just n  -> Var n
    Nothing -> fromMaybe (Var 0) (MultiBimap.lookupR c context)
```

while the translation from a de Bruijn expression to a natural one is done considering an infinite list of possible variable names and keeping a list of currently-on-scope variables to name the indices.

```haskell
-- | An infinite list of all possible variable names
-- in lexicographical order.
variableNames :: [String]
variableNames = concatMap (`replicateM` ['a'..'z']) [1..]

-- | A function translating a deBruijn expression into a
-- natural lambda expression.
nameIndexes :: [String] -> [String] -> Exp -> NamedLambda
nameIndexes _    _   (Var 0) = LambdaVariable "undefined"
nameIndexes used _   (Var n) =
  LambdaVariable (used !! pred (fromInteger n))
nameIndexes used new (Lambda e) =
  LambdaAbstraction (head new) (nameIndexes (head new:used) (tail new) e)
nameIndexes used new (App f g) =
  LambdaApplication (nameIndexes used new f) (nameIndexes used new g)
```

5.5 EVALUATION

As we proved on Corollary 1, the leftmost reduction strategy will find the normal form of any given term provided that it exists. Consequently, we will implement reduction in our interpreter using a function that simply applies the leftmost possible reductions at each step. As a side benefit, this will allow us to show how the interpreter performs step-by-step evaluations to the final user, as discussed in the verbose mode section.

```haskell
-- | Simplifies the expression recursively.
-- Applies only one parallel beta reduction at each step.
simplify :: Exp -> Exp
simplify (Lambda e)           = Lambda (simplify e)
simplify (App (Lambda f) x)   = betared (App (Lambda f) x)
simplify (App (Var e) x)      = App (Var e) (simplify x)
simplify (App a b)            = App (simplify a) (simplify b)
simplify (Var e)              = Var e

-- | Applies repeated simplification to the expression until it stabilizes and
-- returns all the intermediate results.
simplifySteps :: Exp -> [Exp]
simplifySteps e
  | e == s    = [e]
  | otherwise = e : simplifySteps s
  where s = simplify e
```

From the code we can see that the evaluation finishes whenever the expression stabilizes. This can happen in two different cases

- there are no more possible β-reductions, and the algorithm stops; or
- β-reductions do not change the expression. The computation would lead to an infinite loop, so it is immediately stopped. An common example of this is the λ-term $(\lambda x.xx)(\lambda x.xx)$.

5.6 PRINCIPAL TYPE INFERENCE

The interpreter implements the unification and type inference algorithms described in Lemma 4 and Theorem 5. Their recursive nature makes them very easy to implement directly on Haskell. We implement a simply-typed lambda calculus with Curry-style typing and type templates. Our type system has

- an unit type;
- a void type;
- product types;
- union types;
- and function types.

```
-- | A type template is a free type variable or an arrow between two
-- types; that is, the function type.
data Type = Tvar Variable
          | Arrow Type Type
          | Times Type Type
          | Union Type Type
          | Unitty
          | Bottom
          deriving (Eq)
```

We will work with substitutions on type templates. They can be directly defined as functions from types to types. A basic substitution that inserts a given type on the place of a variable will be our building block for more complex ones.

```
type Substitution = Type -> Type

-- | A basic substution. It changes a variable for a type
subs :: Variable -> Type -> Substitution
subs x typ (Tvar y)
  | x == y    = typ
  | otherwise = Tvar y
subs x typ (Arrow a b) = Arrow (subs x typ a) (subs x typ b)
subs x typ (Times a b) = Times (subs x typ a) (subs x typ b)
subs x typ (Union a b) = Union (subs x typ a) (subs x typ b)
subs _ _ Unitty = Unitty
subs _ _ Bottom = Bottom
```

Unification will be implemented making extensive use of the Maybe monad. If the unification fails, it will return an error value, and the error will be propagated to the whole computation. The algorithm is exactly the same that was defined in Lemma 4.

```
-- | Unifies two types with their most general unifier. Returns the substitution
-- that transforms any of the types into the unifier.
unify :: Type -> Type -> Maybe Substitution
unify (Tvar x) (Tvar y)
  | x == y    = Just id
  | otherwise = Just (subs x (Tvar y))
unify (Tvar x) b
  | occurs x b = Nothing
  | otherwise  = Just (subs x b)
unify a (Tvar y)
  | occurs y a = Nothing
  | otherwise  = Just (subs y a)
unify (Arrow a b) (Arrow c d) = unifypair (a,b) (c,d)
unify (Times a b) (Times c d) = unifypair (a,b) (c,d)
unify (Union a b) (Union c d) = unifypair (a,b) (c,d)
unify Unitty Unitty = Just id
unify Bottom Bottom = Just id
unify _ _ = Nothing

-- | Unifies a pair of types
unifypair :: (Type,Type) -> (Type,Type) -> Maybe Substitution
unifypair (a,b) (c,d) = do
  p <- unify b d
  q <- unify (p a) (p c)
  return (q . p)
```

The type inference algorithm is more involved. It takes a list of fresh variables, a type context, a lambda expression and a constraint on the type, expressed as a type template. It outputs a substitution. As an example, the following code shows the type inference algorithm for function types.

```
-- | Type inference algorithm. Infers a type from a given context and expression
-- with a set of constraints represented by a unifier type. The result type must
-- be unifiable with this given type.
typeinfer :: [Variable] -- ^ List of fresh variables
```

```
          -> Context    -- ^ Type context
          -> Exp        -- ^ Lambda expression whose type has to be inferred
          -> Type       -- ^ Constraint
          -> Maybe Substitution

typeinfer (x:vars) ctx (App p q) b = do -- Writing inside the Maybe monad.
  sigma <- typeinfer (evens vars) ctx                  p (Arrow (Tvar x) b)
  tau   <- typeinfer (odds  vars) (applyctx sigma ctx) q (sigma (Tvar x))
  return (tau . sigma)
  where
    -- The list of fresh variables has to be split into two
    odds [] = []
    odds [_] = []
    odds (_:e:xs) = e : odds xs
    evens [] = []
    evens [e] = [e]
    evens (e:_:xs) = e : evens xs
```

The final form of the type inference algorithm will use a normalization algorithm shortening the type names and will apply the type inference to the empty type context. The complete code can be found on the [BROKEN LINK: *Mikrokosmos complete code]. A generalized version of the type inference algorithm is used to generate derivation trees from terms, as it was described in Propositions as types.

In order to draw these diagrams in Unicode characters, a data type for character blocks has been defined. A monoidal structure is defined over them; blocks can be joined vertically and horizontally; and every deduction step can be drawn independently.

```
newtype Block = Block { getBlock :: [String] }
  deriving (Eq, Ord)

instance Monoid Block where
  mappend = joinBlocks -- monoid operation, joins blocks vertically
  mempty  = Block [[]] -- neutral element

-- Type signatures
joinBlocks :: Block -> Block -> Block
stackBlocks :: String -> Block -> Block -> Block
textBlock :: String -> Block
deductionBlock :: Block -> String -> [Block] -> Block
box :: Block -> Block
```

6 USER INTERACTION

6.1 MONADIC PARSER COMBINATORS

A common approach to building parsers in functional programming is to model parsers as functions. Higher-order functions on parsers act as *combinators*, which are used to implement complex parsers in a modular way from a set of primitive ones. In this setting, parsers exhibit a monad algebraic structure, which can be used to simplify the combination of parsers. A technical report on **monadic parser combinators** can be found on [HM96].

The use of monads for parsing was discussed for the first time in [Wad85], and later in [Wad90] and [HM98]. The parser type is defined as a function taking an input `String` and returning a list of pairs, representing a successful parse each. The first component of the pair is the parsed value and the second component is the remaining input. The Haskell code for this definition is

```haskell
-- A parser takes a string an returns a list of possible parsings with
-- their remaining string.
newtype Parser a = Parser (String -> [(a,String)])
parse :: Parser a -> String -> [(a,String)]
parse (Parser p) = p
-- A parser can be composed monadically, the composed parser (p >>= q)
-- applies q to every possible parsing of p. A trivial one is defined.
instance Monad Parser where
  return x = Parser (\s -> [(x,s)]) -- Trivial parser, directly returns x.
  p >>= q  = Parser (\s -> concat [parse (q x) s' | (x,s') <- parse p s ])
```

where the monadic structure is defined by `bind` and `return`. Given a value, the `return` function creates a parser that consumes no input and simply returns the given value. The »= function acts as a sequencing operator for parsers. It takes two parsers and applies the second one over the remaining inputs of the first one, using the parsed values on the first parsing as arguments.

An example of primitive **parser** is the `item` parser, which consumes a character from a non-empty string. It is written in Haskell code using pattern matching on the string as

```
item :: Parser Char
item = Parser (\s -> case s of "" -> []; (c:s') -> [(c,s')])
```

and an example of **parser combinator** is the `many` function, which creates a parser that allows one or more applications of the given parser

```
many :: Paser a -> Parser [a]
many p = do
  a  <- p
  as <- many p
  return (a:as)
```

in this example `many item` would be a parser consuming all characters from the input string.

6.2 PARSEC

Parsec is a monadic parser combinator Haskell library described in [Lei01]. We have chosen to use it due to its simplicity and extensive documentation. As we expect to use it to parse user live input, which will tend to be short, performance is not a critical concern. A high-performace library supporting incremental parsing, such as **Attoparsec** [O'S16], would be suitable otherwise.

6.3 VERBOSE MODE

As we explained previously on the evaluation section, the simplification can be analyzed step-by-step. The interpreter allows us to see the complete evaluation when the `verbose` mode is activated. To activate it, we can execute `:verbose on` in the interpreter.

The difference can be seen on the following example, which shows the execution of $1 + 2$, first without intermediate results, and later, showing every intermediate step.

```
mikro> plus 1 2
λa.λb.(a (a (a b))) ⇒ 3

mikro> :verbose on
verbose: on
mikro> plus 1 2
```

```
((plus 1) 2)
((λλλλ((4 2) ((3 2) 1)) λλ(2 1)) λλ(2 (2 1)))
(λλλ((λλ(2 1) 2) ((3 2) 1)) λλ(2 (2 1)))
λλ((λλ(2 1) 2) ((λλ(2 (2 1)) 2) 1))
λλ(λ(3 1) (λ(3 (3 1)) 1))
λλ(2 (λ(3 (3 1)) 1))
λλ(2 (2 (2 1)))

λa.λb.(a (a (a b))) ⇒ 3
```

The interpreter output can be colored to show specifically where it is performing reductions. It is activated by default, but can be deactivated by executing `:color off`. The following code implements *verbose mode* in both cases.

```haskell
-- | Shows an expression, coloring the next reduction if necessary
showReduction :: Exp -> String
showReduction (Lambda e)         = "λ" ++ showReduction e
showReduction (App (Lambda f) x) = betaColor (App (Lambda f) x)
showReduction (Var e)            = show e
showReduction (App rs x)         =
  "(" ++ showReduction rs ++ " " ++ showReduction x ++ ")"
showReduction e                  = show e
```

6.4 SKI MODE

Every λ-term can be written in terms of SKI combinators. SKI combinator expressions can be defined as a binary tree having S, K, and I as possible leafs.

```haskell
data Ski = S | K | I | Comb Ski Ski | Cte String
```

The SKI-abstraction and bracket abstraction algorithms are implemented on Mikrokosmos, and they can be used by activating the *ski mode* with `:ski on`. When this mode is activated, every result is written in terms of SKI combinators.

```
mikro> 2
λa.λb.(a (a b)) ⇒ S(S(KS)K)I ⇒ 2
mikro> and
λa.λb.((a b) a) ⇒ SSK ⇒ and
```

The code implementing these algorithms follows directly from the theoretical version in [HS08].

```haskell
-- | Bracket abstraction of a SKI term, as defined in Hindley-Seldin
-- (2.18).
```

```
bracketabs :: String -> Ski -> Ski
bracketabs x (Cte y) = if x == y then I else Comb K (Cte y)
bracketabs x (Comb u (Cte y))
  | freein x u && x == y = u
  | freein x u           = Comb K (Comb u (Cte y))
  | otherwise            = Comb (Comb S (bracketabs x u)) (bracketabs x (Cte y))
bracketabs x (Comb u v)
  | freein x (Comb u v)  = Comb K (Comb u v)
  | otherwise            = Comb (Comb S (bracketabs x u)) (bracketabs x v)
bracketabs _ a           = Comb K a


-- | SKI abstraction of a named lambda term. From a lambda expression
-- creates a SKI equivalent expression. The following algorithm is a
-- version of the algorithm (9.10) on the Hindley-Seldin book.
skiabs :: NamedLambda -> Ski
skiabs (LambdaVariable x)      = Cte x
skiabs (LambdaApplication m n) = Comb (skiabs m) (skiabs n)
skiabs (LambdaAbstraction x m) = bracketabs x (skiabs m)
```

7

USAGE

7.1 INSTALLATION

The complete Mikrokosmos suite is divided in multiple parts:

1. the **Mikrokosmos interpreter**, written in Haskell;
2. the **Jupyter kernel**, written in Python;
3. the **CodeMirror Lexer**, written in Javascript;
4. the **Mikrokosmos libraries**, written in the Mikrokosmos language;
5. the **Mikrokosmos-js** compilation, which can be used in web browsers.

These parts will be detailed on the following sections. A system that already satisfies all dependencies (Stack, Pip and Jupyter), can install Mikrokosmos using the following script, which is detailed on this section

```
# Mikrokosmos interpreter
stack install mikrokosmos
# Jupyter kernel for Mikrokosmos
sudo pip install imikrokosmos
# Libraries
git clone https://github.com/M42/mikrokosmos-lib.git ~/.mikrokosmos
```

The **Mikrokosmos interpreter** is listed in the central Haskell package archive, *Hackage* [1]. The packaging of Mikrokosmos has been done using the **cabal** tool; and the configuration of the package can be read in the file `mikrokosmos.cabal` on the Mikrokosmos code. As a result, Mikrokosmos can be installed using the **cabal** and **stack** Haskell package managers. That is,

```
# With cabal
cabal install mikrokosmos
```

[1] : Hackage can be accesed in: http://hackage.haskell.org/ and the Mikrokosmos package can be found in https://hackage.haskell.org/package/mikrokosmos

```
# With stack
stack install mikrokosmos
```

The **Mikrokosmos Jupyter kernel** is listed in the central Python package archive. Jupyter is a dependency of this kernel, which only can be used in conjunction with it. It can be installed with the `pip` package manager as

```
sudo pip install imikrokosmos
```

and the installation can be checked by listing the available Jupyter kernels with

```
jupyter kernelspec list
```

The **Mikrokosmos libraries** can be downloaded directly from its GitHub repository. [2] They have to be placed under `~/.mikrokosmos` if we want them to be locally available or under `/usr/lib/mikrokosmos` if we want them to be globally available.

```
git clone https://github.com/M42/mikrokosmos-lib.git ~/.mikrokosmos
```

The following script installs the complete Mikrokosmos suite on a fresh system. It has been tested under `Ubuntu 16.04.3 LTS (Xenial Xerus)`.

```
# 1. Installs Stack, the Haskell package manager
wget -qO- https://get.haskellstack.org | sh
STACK=$(which stack)

# 2. Installs the ncurses library, used by the console interface
sudo apt install libncurses5-dev libncursesw5-dev

# 3. Installs the Mikrokosmos interpreter using Stack
$STACK setup
$STACK install mikrokosmos

# 4. Installs the Mikrokosmos standard libraries
sudo apt install git
git clone https://github.com/M42/mikrokosmos-lib.git ~/.mikrokosmos

# 5. Installs the IMikrokosmos kernel for Jupyter
sudo apt install python3-pip
sudo -H pip install --upgrade pip
sudo -H pip install jupyter
sudo -H pip install imikrokosmos
```

[2] : The repository can be accessed in: `https://github.com/M42/mikrokosmos-lib.git`

7.2 mikrokosmos interpreter

Once installed, the Mikrokosmos λ interpreter can be opened from the terminal with the `mikrokosmos` command. It will enter a *read-eval-print loop* where λ-expressions and interpreter commands can be evaluated.

```
$> mikrokosmos
Welcome to the Mikrokosmos Lambda Interpreter!
Version 0.5.0. GNU General Public License Version 3.
mikro> _
```

The interpreter evaluates every line as a lambda expression. Examples on the use of the interpreter can be read on the following sections. Apart from the evaluation of expressions, the interpreter accepts the following commands

- `:quit` and `:restart`, stop the interpreter;
- `:verbose` activates *verbose mode*;
- `:ski` activates *SKI mode*;
- `:types` changes between untyped and simply typed λ-calculus;
- `:color` deactivates colored output;
- `:load` loads a library.

Figure 2 is an example session on the mikrokosmos interpreter.

7.3 jupyter kernel

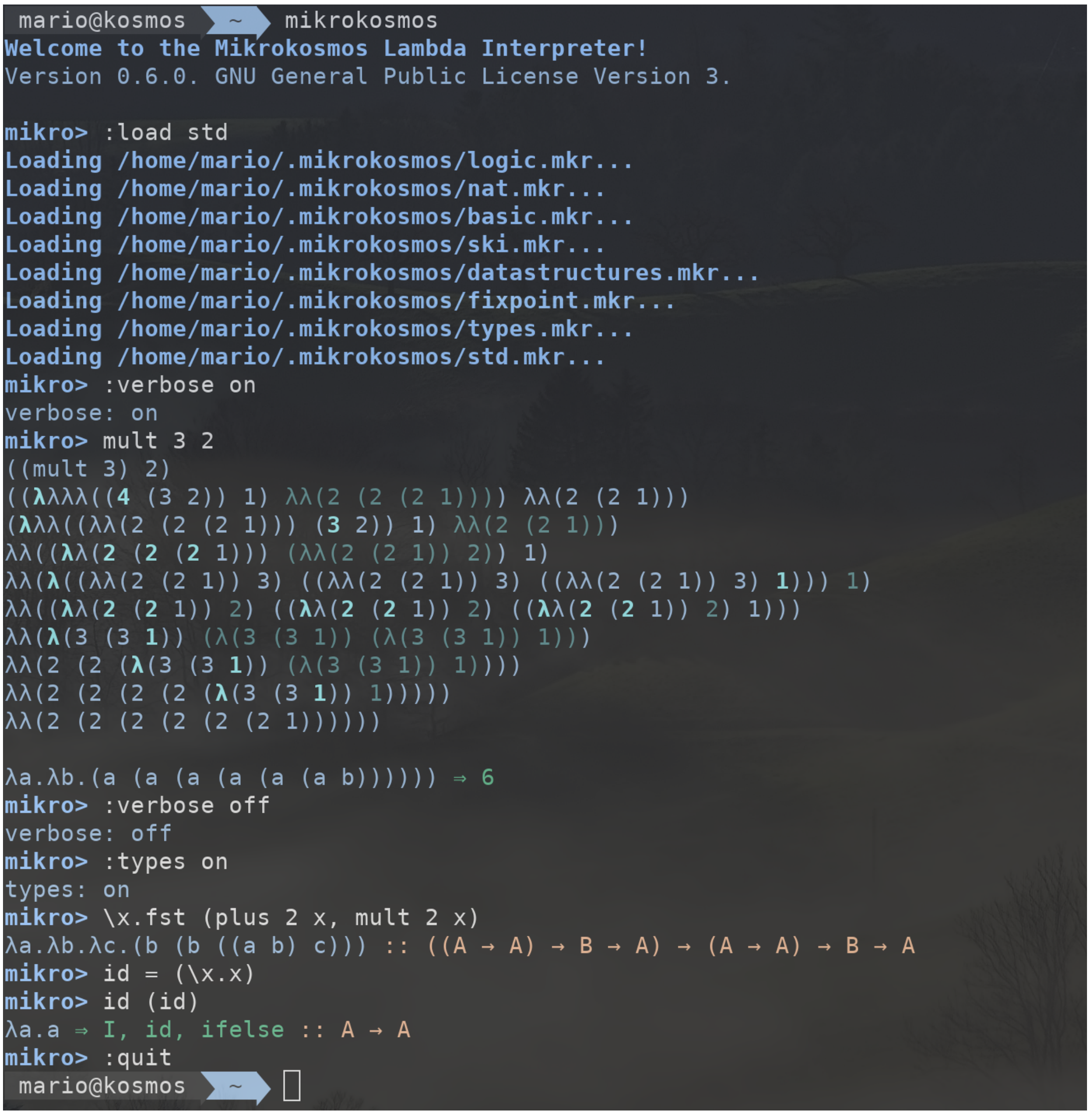
The **Jupyter Project** [Jup] is an open source project providing support for interactive scientific computing. Specifically, the Jupyter Notebook provides a web application for creating interactive documents with live code and visualizations.

We have developed a Mikrokosmos kernel for the Jupyter Notebook, allowing the user to write and execute arbitrary Mikrokosmos code on this web application. An example session can be seen on Figure 3.

The implementation is based on the `pexpect` library for Python. It allows direct interaction with any REPL and collects its results. Specifically, the following Python lines represent the central idea of this implementation

```
# Initialization
mikro = pexpect.spawn('mikrokosmos')
mikro.expect('mikro>')

# Interpreter interaction
# Multiple-line support
output = ""
```

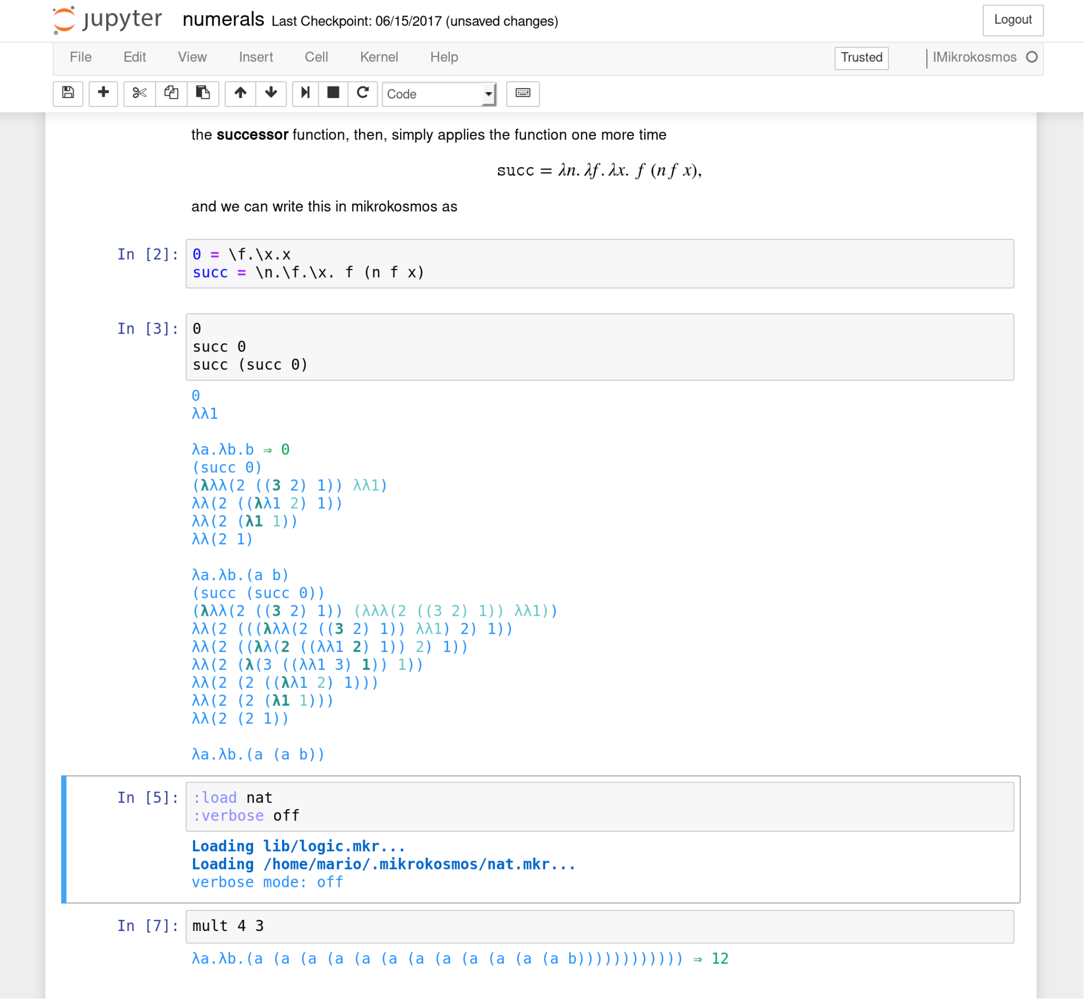


```
 mario@kosmos  ~  mikrokosmos
Welcome to the Mikrokosmos Lambda Interpreter!
Version 0.6.0. GNU General Public License Version 3.

mikro> :load std
Loading /home/mario/.mikrokosmos/logic.mkr...
Loading /home/mario/.mikrokosmos/nat.mkr...
Loading /home/mario/.mikrokosmos/basic.mkr...
Loading /home/mario/.mikrokosmos/ski.mkr...
Loading /home/mario/.mikrokosmos/datastructures.mkr...
Loading /home/mario/.mikrokosmos/fixpoint.mkr...
Loading /home/mario/.mikrokosmos/types.mkr...
Loading /home/mario/.mikrokosmos/std.mkr...
mikro> :verbose on
verbose: on
mikro> mult 3 2
((mult 3) 2)
((λλλλ((4 (3 2)) 1) λλ(2 (2 (2 1)))) λλ(2 (2 1)))
(λλλ((λλ(2 (2 (2 1))) (3 2)) 1) λλ(2 (2 1)))
λλ((λλ(2 (2 (2 1))) (λλ(2 (2 1)) 2)) 1)
λλ(λ((λλ(2 (2 1)) 3) ((λλ(2 (2 1)) 3) ((λλ(2 (2 1)) 3) 1))) 1)
λλ((λλ(2 (2 1)) 2) ((λλ(2 (2 1)) 2) ((λλ(2 (2 1)) 2) 1)))
λλ(λ(3 (3 1)) (λ(3 (3 1)) (λ(3 (3 1)) 1)))
λλ(2 (2 (λ(3 (3 1)) (λ(3 (3 1)) 1))))
λλ(2 (2 (2 (2 (λ(3 (3 1)) 1)))))
λλ(2 (2 (2 (2 (2 (2 1))))))

λa.λb.(a (a (a (a (a (a b)))))) ⇒ 6
mikro> :verbose off
verbose: off
mikro> :types on
types: on
mikro> \x.fst (plus 2 x, mult 2 x)
λa.λb.λc.(b (b ((a b) c))) :: ((A → A) → B → A) → (A → A) → B → A
mikro> id = (\x.x)
mikro> id (id)
λa.a ⇒ I, id, ifelse :: A → A
mikro> :quit
 mario@kosmos  ~  ▯
```

Figure 2: Mikrokosmos interpreter session.



the **successor** function, then, simply applies the function one more time

$$\mathtt{succ} = \lambda n.\lambda f.\lambda x.\ f\ (n\ f\ x),$$

and we can write this in mikrokosmos as

```
In [2]: 0 = \f.\x.x
        succ = \n.\f.\x. f (n f x)
```

```
In [3]: 0
        succ 0
        succ (succ 0)

        0
        λλ1

        λa.λb.b ⇒ 0
        (succ 0)
        (λλλ(2 ((3 2) 1)) λλ1)
        λλ(2 ((λλ1 2) 1))
        λλ(2 (λ1 1))
        λλ(2 1)

        λa.λb.(a b)
        (succ (succ 0))
        (λλλ(2 ((3 2) 1)) (λλλ(2 ((3 2) 1)) λλ1))
        λλ(2 (((λλλ(2 ((3 2) 1)) λλ1) 2) 1))
        λλ(2 ((λλ(2 ((λλ1 2) 1)) 2) 1))
        λλ(2 (λ(3 ((λλ1 3) 1)) 1))
        λλ(2 (2 ((λλ1 2) 1)))
        λλ(2 (2 (λ1 1)))
        λλ(2 (2 1))

        λa.λb.(a (a b))
```

```
In [5]: :load nat
        :verbose off

        Loading lib/logic.mkr...
        Loading /home/mario/.mikrokosmos/nat.mkr...
        verbose mode: off
```

```
In [7]: mult 4 3

        λa.λb.(a (a (a (a (a (a (a (a (a (a (a (a b)))))))))))) ⇒ 12
```

Figure 3: Jupyter notebook Mikrokosmos session.

```
for line in code.split('\n'):
    # Send code to mikrokosmos
    self.mikro.sendline(line)
    self.mikro.expect('mikro> ')

    # Receive and filter output from mikrokosmos
    partialoutput = self.mikro.before
    partialoutput = partialoutput.decode('utf8')
    output = output + partialoutput
```

A `pip` installable package has been created following the Python Packaging Authority guidelines. [3] This allows the kernel to be installed directly using the `pip` python package manager.

```
sudo -H pip install imikrokosmos
```

7.4 CODEMIRROR LEXER

CodeMirror [4] is a text editor for the browser implemented in Javascript. It is used internally by the Jupyter Notebook.

A CodeMirror lexer for Mikrokosmos has been written. It uses Javascript regular expressions and signals the ocurrence of any kind of operator to CodeMirror. It enables syntax highlighting for Mikrokosmos code on Jupyter Notebooks. It comes bundled with the kernel specification and no additional installation is required.

```
CodeMirror.defineSimpleMode("mikrokosmos", {
    start: [
    // Comments
    {regex: /\#.*/,
    token: "comment"},
    // Interpreter
    {regex: /\:load|\:verbose|\:ski|\:restart|\:types|\:color/,
    token: "atom"},
    // Binding
    {regex: /(.*?)(\s*)(=)(\s*)(.*?)$/,
    token: ["def",null,"operator",null,"variable"]},
    // Operators
    {regex: /[=!]+/,
    token: "operator"},
    ],
```

[3] The PyPA packaging user guide can be found in its official page: https://packaging.python.org/

[4] Documentation for CodeMirror can be found in its official page: https://codemirror.net/

```
    meta: {
        dontIndentStates: ["comment"],
        lineComment: "#"
    }
}
```

7.5 JUPYTERHUB

JupyterHub manages multiple instances of independent single-user Jupyter notebooks. We used it to serve Mikrokosmos notebooks and tutorials to students studying λ-calculus.

In order to install Mikrokosmos on a server and use it as `root` user, we need

- to clone the libraries into `/usr/lib/mikrokosmos`. They should be available system-wide.
- to install the Mikrokosmos interpreter into `/usr/local/bin`. In this case, we chose not to install Mikrokosmos from source, but simply copy the binaries and check the availability of the `ncurses` library.
- to install the Mikrokosmos Jupyter kernel as usual.

Our server used a SSL certificate; and OAuth autentication via GitHub. Mikrokosmos tutorials were installed for every student.

7.6 CALLING MIKROKOSMOS FROM JAVASCRIPT

The GHCjs[5] compiler allows transpiling from Haskell to Javascript. Its foreign function interface allows a Haskell function to be passed as a continuation to a Javascript function.

A particular version of the `Main.hs` module of Mikrokosmos was written in order to provide a `mikrokosmos` function, callable from Javascript. This version includes the standard libraries automatically and reads blocks of texts as independent Mikrokosmos commands. The relevant use of the foreign function interface is showed in the following code

```
foreign import javascript unsafe "mikrokosmos = $1"
    set_mikrokosmos :: Callback a -> IO ()
```

which provides `mikrokosmos` as a Javascript function once the code is transpiled. In particular, the following is an example of how to call Mikrokosmos from Javascript

[5] : The GHCjs documentation is available on its web page https://github.com/ghcjs/ghcjs

```
button.onclick = function () {
   editor.save();
   outputcode.getDoc().setValue(mikrokosmos(inputarea.value).mkroutput);
   textAreaAdjust(outputarea);
}
```

A small script has been written in Javascript to help with the task of embedding Mikrokosmos into a web page. It and can be included directly from

`https://m42.github.io/mikrokosmos-js/mikrobox.js`

using GitHub as a CDN. It will convert any HTML script tag written as follows

```
<div class="mikrojs-console">
<script type="text/mikrokosmos">
(λx.x)
... your code
</script>
</div>
```

into a CodeMirror pad where Mikrokosmos can be executed. The Mikrokosmos tutorials are an example of this feature and can be seen on Figure 4.

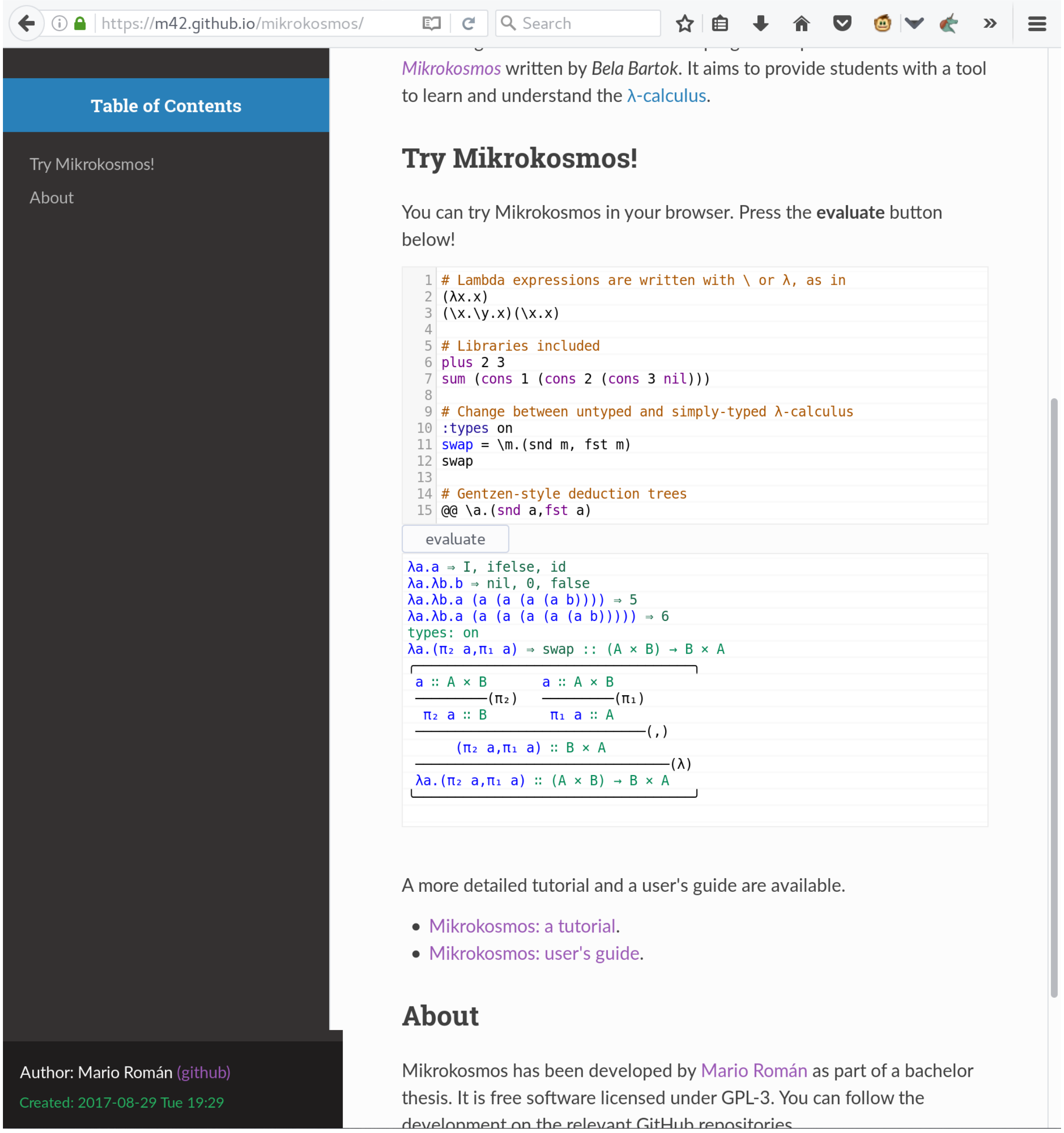


Table of Contents

Try Mikrokosmos!

About

Mikrokosmos written by *Bela Bartok*. It aims to provide students with a tool to learn and understand the λ-calculus.

Try Mikrokosmos!

You can try Mikrokosmos in your browser. Press the **evaluate** button below!

```
# Lambda expressions are written with \ or λ, as in
(λx.x)
(\x.\y.x)(\x.x)

# Libraries included
plus 2 3
sum (cons 1 (cons 2 (cons 3 nil)))

# Change between untyped and simply-typed λ-calculus
:types on
swap = \m.(snd m, fst m)
swap

# Gentzen-style deduction trees
@@ \a.(snd a,fst a)
```

```
λa.a ⇒ I, ifelse, id
λa.λb.b ⇒ nil, 0, false
λa.λb.a (a (a (a (a b)))) ⇒ 5
λa.λb.a (a (a (a (a (a b))))) ⇒ 6
types: on
λa.(π₂ a,π₁ a) ⇒ swap :: (A × B) → B × A
┌────────────────────────────────────┐
 a :: A × B       a :: A × B
 ──────────(π₂)   ──────────(π₁)
  π₂ a :: B        π₁ a :: A
 ─────────────────────────────(,)
      (π₂ a,π₁ a) :: B × A
 ────────────────────────────────(λ)
 λa.(π₂ a,π₁ a) :: (A × B) → B × A
└────────────────────────────────────┘
```

A more detailed tutorial and a user's guide are available.

- Mikrokosmos: a tutorial.
- Mikrokosmos: user's guide.

About

Mikrokosmos has been developed by Mario Román as part of a bachelor thesis. It is free software licensed under GPL-3. You can follow the development on the relevant GitHub repositories.

Author: Mario Román (github)

Created: 2017-08-29 Tue 19:29

Figure 4: Mikrokosmos embedded into a web page.

8

PROGRAMMING ENVIRONMENT

8.1 CABAL, STACK AND HADDOCK

The Mikrokosmos documentation as a Haskell library is included in its own code. It uses **Haddock**, a tool that generates documentation from annotated Haskell code; it is the *de facto* standard for Haskell software.

Dependencies and packaging details for Mikrokosmos are specified in a file distributed with the source code called `mikrokosmos.cabal`. It is used by the package managers **stack** and **cabal** to provide the necessary libraries even if they are not available system-wide. The **stack** tool is also used to package the software, which is uploaded to ***Hackage***.

8.2 TESTING

Tasty is the Haskell testing framework of our choice for this project. It allows the user to create a comprehensive test suite combining multiple types of tests. The Mikrokosmos code is tested using the following techniques

- **unit tests**, in which individual core functions are tested independently of the rest of the application;
- **property-based testing**, in which multiple test cases are created automatically in order to verfiy that a specified property always holds;
- **golden tests**, a special case of unit tests in which the expected results of an IO action, as described on a file, are checked to match the actual ones.

We are using the **HUnit** library for unit tests. It tests particular cases of type inference, unification and parsing. The following is an example of unit test, as found in `tests.hs`. It checks that the type inference of the identity term is correct.

```
-- Checks that the type of λx.x is exactly A → A
testCase "Identity type inference" $
  typeinference (Lambda (Var 1)) @?= Just (Arrow (Tvar 0) (Tvar 0))
```

We are using the **QuickCheck** library for property-based tests. It tests transformation properties of lambda expressions. In the following example, it tests that any De Bruijn expression keeps its meaning when translated into a λ-term.

```
-- Tests if translation preserves meaning
QC.testProperty "Expression -> named -> expression" $
  \expr -> toBruijn emptyContext (nameExp expr) == expr
```

We are using the **tasty-golden** package for golden tests. Mikrokosmos can be passed a file as an argument to interpret it and show only the results. This feature is used to create a golden test in which the interpreter is asked to provide the correct interpretation of a given file. This file is called `testing.mkr`, and contains library definitions and multiple tests. Its expected output is `testing.golden`. For example, the following Mikrokosmos code can be found on the file

```
:types on
caseof (inr 3) (plus 2) (mult 2)
```

and the expected output is

```
-- types: on
-- λa.λb.(a (a (a (a (a (a b)))))) ⇒ 6 :: (A → A) → A → A
```

8.3 VERSION CONTROL AND CONTINUOUS INTEGRATION

Mikrokosmos uses **git** as its version control system and the code, which is licensed under GPLv3, can be publicly accessed on the following GitHub repository:

https://github.com/M42/mikrokosmos

Development takes place on the `development` git branch and permanent changes are released into the `master` branch. Some more minor repositories have been used in the development; they directly depend on the main one

- https://github.com/m42/mikrokosmos-js
- https://github.com/M42/jupyter-mikrokosmos
- https://github.com/M42/mikrokosmos-lib

The code uses the **Travis CI** continuous integration system to run tests and check that the software builds correctly after each change and in a reproducible way on a fresh Linux installation provided by the service.

9

PROGRAMMING IN UNTYPED λ-CALCULUS

This section explains how to use untyped λ-calculus to encode data structures and useful data, such as booleans, linked lists, natural numbers or binary trees. All this is done on pure λ-calculus avoiding the addition of new syntax or axioms.

This presentation follows the Mikrokosmos tutorial on λ-calculus, which aims to teach how it is possible to program using untyped λ-calculus without discussing technical topics such as those we have discussed on the chapter on untyped λ-calculus. It also follows the exposition on [Sel13] of the usual Church encodings.

All the code on this section is valid Mikrokosmos code.

9.1 BASIC SYNTAX

In the interpreter, λ-abstractions are written with the symbol \, representing a λ. This is a convention used on some functional languages such as Haskell or Agda. Any alphanumeric string can be a variable and can be defined to represent a particular λ-term using the = operator.

As a first example, we define the identity function (`id`), function composition (`compose`) and a constant function on two arguments which always returns the first one untouched (`const`).

```
id = \x.x
compose = \f.\g.\x.f (g x)
const = \x.\y.x
```

Evaluation of terms will be presented as comments to the code,

```
compose id id
-- [1]: λa.a ⇒ id
```

It is important to notice that multiple argument functions are defined as higher one-argument functions that return another functions as arguments. These intermediate functions are also valid λ-terms. For example

```
discard = const id
```

is a function that discards one argument and returns the identity, `id`. This way of defining multiple argument functions is called the **currying** of a function in honor to the american logician Haskell Curry in [CF58]. It is a particular instance of a deeper fact: exponentials are defined by the following adjunction

$$\hom(A \times B, C) \cong \hom(A, \hom(B, C)).$$

9.2 A TECHNIQUE ON INDUCTIVE DATA ENCODING

We will implicitly use a technique on the majority of our data encodings that allows us to write an encoding for any algebraically inductive generated data. This technique is used without comment on [Sel13] and represents the basis of what is called the **Church encoding** of data in λ-calculus.

We start considering the usual inductive representation of a data type with constructors, as we do when representing a syntax with a BNF, for example,

$$\mathtt{Nat} ::= \mathtt{Zero} \mid \mathtt{Succ}\ \mathtt{Nat}.$$

Or, in general

$$\mathtt{D} ::= C_1 \mid C_2 \mid C_3 \mid \dots$$

It is not possible to directly encode constructors on λ-calculus. Even if we were able, they would have, in theory, no computational content; the data structure would not be reduced under any λ-term, and we would need at least the ability to pattern match on the constructors to define functions on them. Our λ-calculus would need to be extended with additional syntax for every new data structure.

This technique, instead, defines a data term as a function on multiple arguments representing the missing constructors. In our example, the number 2, which would be written as $\mathtt{Succ}(\mathtt{Succ}(\mathtt{Zero}))$, would be encoded as

$$2 = \lambda s.\ \lambda z.\ s(s(z)).$$

In general, any instance of the data structure `D` would be encoded as a λ-expression depending on all its constuctors

$$\lambda c_1.\ \lambda c_2.\ \lambda c_3.\ \dots\ \lambda c_n.(term).$$

This acts as the definition of an initial algebra over the constructors and lets us to compute over instances of that algebra by instantiating it on particular cases. Particular examples are described on the following sections.

9.3 BOOLEANS

Booleans can be defined as the data generated by a pair of constuctors

$$\texttt{Bool} ::= \texttt{True} \mid \texttt{False}.$$

Consequently, the Church encoding of booleans takes these constructors as arguments and defines

```
true  = \t.\f.t
false = \t.\f.f
```

Note that `true` and `const` are exactly the same term up to α-conversion. The same thing happens with `false` and `alwaysid`. The absence of types prevents us to make any effort to discriminate between these two uses of the same λ-term. Another side-effect of this definition is that our `true` and `false` terms can be interpreted as binary functions choosing between two arguments, that is,

- $\texttt{true}(a,b) = a$,
- $\texttt{false}(a,b) = b$.

We can test this interpretation on the interpreter to get

```
true id const
false id const
--- [1]: id
--- [2]: const
```

This inspires the definition of an `ifelse` combinator as the identity

```
ifelse = \b.b
(ifelse true) id const
(ifelse false) id const
--- [1]: id
--- [2]: const
```

The usual logic gates can be defined profiting from this interpretation of booleans

```
and = \p.\q.p q p
or = \p.\q.p p q
```

```
not = \b.b false true
xor = \a.\b.a (not b) b
implies = \p.\q.or (not p) q

xor true true
and true true
--- [1]: false
--- [2]: true
```

9.4 natural numbers

Our definition of natural numbers is inspired by the Peano natural numbers. We use two constructors

- zero is a natural number, written as Z;
- the successor of a natural number is a natural number, written as S;

and the BNF we defined when discussing how to encode inductive data.

```
0    = \s.\z.z
succ = \n.\s.\z.s (n s z)
```

This definition of `0` is trivial: given a successor function and a zero, return zero. The successor function seems more complex, but it uses the same underlying idea: given a number, a successor and a zero, apply the successor to the interpretation of that number using the same successor and zero.

We can then name some natural numbers as

```
1 = succ 0
2 = succ 1
3 = succ 2
4 = succ 3
5 = succ 4
6 = succ 5
...
```

even if we can not define an infinite number of terms as we might wish. The interpretation the natural number *n* as a higher order function is a function taking an argument `f` and applying them *n* times over the second argument.

```
5 not true
4 not true
double = \n.\s.\z.n (compose s s) z
double 3
```

```
--- [1]: false
--- [2]: true
--- [3]: 6
```

Addition $n + m$ applies the successor m times to n; and multiplication nm applies the n-fold application of the successor m times to 0.

```
plus = \m.\n.\s.\z.m s (n s z)
mult = \m.\n.\s.\z.m (n s) z
plus 2 1
mult 2 4
--- [1]: 3
--- [2]: 8
```

9.5 the predecessor function and predicates on numbers

The predecessor function is much more complex than the previous ones. As we can see, it is not trivial how could we compute the predecessor using the limited form of induction that Church numerals allow.

Stephen Kleene, one of the students of Alonzo Church only discovered how to write the predecessor function after thinking about it for a long time (and he only discovered it while a long visit at the dentist's, which is the reason why this definition is often called the ***wisdom tooth trick***, see [Cro75]). We will use a slightly different version of the definition that does not depend on a pair datatype.

We will start defining a *reverse composition* operator, called `rcomp`; and we will study what happens when it is composed to itself; that is

```
rcomp = \f.\g.\h.h (g f)
\f.3 (inc f)
\f.4 (inc f)
\f.5 (inc f)
--- [1]: λa.λb.λc.c (a (a (b a)))
--- [2]: λa.λb.λc.c (a (a (a (b a))))
--- [3]: λa.λb.λc.c (a (a (a (a (b a)))))
```

will allow us now to use the `b` argument to discard the first instance of the `a` argument and return the same number wihtout the last constructor. Thus, our definition of `pred` is

```
pred = \n.\s.\z.(n (inc s) (\x.z) (\x.x))
```

From the definition of `pred`, some predicates on numbers can be defined. The first predicate will be a function distinguishing a successor from a zero. It will be user later to build more complex ones. It is built by appliying a `const false` function `n` times to a true constant. Only if it is applied `0` times, it will return a true value.

```
iszero = \n.(n (const false) true)
iszero 0
iszero 2
--- [1]: true
--- [2]: false
```

From this predicate, we can derive predicates on equality and ordering.

```
leq = \m.\n.(iszero (minus m n))
eq  = \m.\n.(and (leq m n) (leq n m))
```

9.6 LISTS AND TREES

We would need two constructors to represent a list: a `nil` signaling the end of the list and a `cons`, joining an element to the head of the list. An example of list would be

$$\texttt{cons}\ 1\ (\texttt{cons}\ 2\ (\texttt{cons}\ 3\ \texttt{nil})).$$

Our definition takes those two constructors into account

```
nil  = \c.\n.n
cons = \h.\t.\c.\n.(c h (t c n))
```

and the interpretation of a list as a higher-order function is its `fold` function, a function taking a binary operation and an initial element and appliying the operation repeteadly to every element on the list.

$$\texttt{cons}\ 1\ (\texttt{cons}\ 2\ (\texttt{cons}\ 3\ \texttt{nil}))\xrightarrow{fold\ plus\ 0}\texttt{plus}\ 1\ (\texttt{plus}\ 2\ (\texttt{plus}\ 3\ 0)) = 6$$

The `fold` operation and some operations on lists can be defined explicitly as

```
fold = \c.\n.\l.(l c n)
sum  = fold plus 0
prod = fold mult 1
all  = fold and true
any  = fold or false
length = foldr (\h.\t.succ t) 0
```

```
sum (cons 1 (cons 2 (cons 3 nil)))
all (cons true (cons true (cons true nil)))
--- [1]: 6
--- [2]: true
```

The two most commonly used particular cases of fold and frequent examples of the functional programming paradigm are `map` and `filter`.

- The **map** function applies a function `f` to every element on a list.
- The **filter** function removes the elements of the list that do not satisfy a given predicate. It *filters* the list, leaving only elements that satisfy the predicate.

They can be defined as follows.

```
map    = \f.(fold (\h.\t.cons (f h) t) nil)
filter = \p.(foldr (\h.\t.((p h) (cons h t) t)) nil)
```

On `map`, given a `cons h t`, we return a `cons (f h) t`; and given a `nil`, we return a `nil`. On `filter`, we use a boolean to decide at each step whether to return a list with a head or return the tail ignoring the head.

```
mylist = cons 1 (cons 2 (cons 3 nil))
sum (map succ mylist)
length (filter (leq 2) mylist)
--- [1]: 9
--- [2]: 2
```

Lists have been defined using two constructors and **binary trees** will be defined using the same technique. The only difference with lists is that the `cons` constructor is replaced by a `node` constructor, which takes two binary trees as arguments. That is, a binary tree is

- an empty tree; or
- a node, containing a label, a left subtree, and a right subtree.

Defining functions using a fold-like combinator is again very simple due to the chosen representation. We need a variant of the usual function acting on three arguments, the label, the right node and the left node.

```
-- Binary tree definition
node = \x.\l.\r.\f.\n.(f x (l f n) (r f n))
-- Example on natural numbers
mytree    = node 4 (node 2 nil nil) (node 3 nil nil)
triplesum = \a.\b.\c.plus (plus a b) c
mytree triplesum 0
--- [1]: 9
```

9.7 FIXED POINTS

A fixpoint combinator is a term representing a higher-order function that, given any function f, solves the equation

$$\mathtt{x} = \mathtt{f}\ \mathtt{x}$$

for x, meaning that, if fix f is the fixpoint of f, the following sequence of equations holds

$$\mathtt{fix}\ f = f(\mathtt{fix}\ f) = f(f(\mathtt{fix}\ f)) = f(f(f(\mathtt{fix}\ f))) = \dots$$

Such a combinator actually exists; it can be defined and used as

```
fix := (\f.(\x.f (x x)) (\x.f (x x)))
fix (const id)
--- [1]: id
```

Where := defines a function without trying to evaluate it to a normal form; this is useful in cases like the previous one, where the function has no normal form. Examples of its applications are a *factorial* function or a *fibonacci* function, as in

```
fact := fix (\f.\n.iszero n 1 (mult n (f (pred n))))
fib  := fix (\f.\n.iszero n 1 (plus (f (pred n)) (f (pred (pred n)))))
fact 3
fib 3
--- [1]: 6
--- [2]: 5
```

Note the use of iszero to stop the recursion.

The fix function cannot be evaluated without arguments into a closed form, so we have to delay the evaluation of the expression when we bind it using !=. Our evaluation strategy, however, will always find a way to reduce the term if it is possible, as we saw in Corollary 1; even if it has intermediate irreducible terms.

```
fix                -- diverges
true  id fix       -- evaluates to id
false id fix       -- diverges
```

Other examples of the interpreter dealing with non terminating functions include infinite lists as in the following examples, where we take the first term of an infinite list without having to evaluate it completely or compare an infinite number arising as the fix point of the successor function with a finite number.

```
-- Head of an infinite list of zeroes
head = fold const false
head (fix (cons 0))
-- Compare infinity with other numbers
infinity != fix succ
leq infinity 6
---- [1]: 0
---- [2]: false
```

These definitions unfold as

- $\mathtt{fix}\ (\mathtt{cons}\ 0) = \mathtt{cons}\ 0\ (\mathtt{cons}\ 0\ (\mathtt{cons}\ 0\ \ldots))$, an infinite list of zeroes;
- $\mathtt{fix}\ \mathtt{succ} \qquad = \mathtt{succ}\ (\mathtt{succ}\ (\mathtt{succ}\ \ldots))$, an infinite natural number.

10

PROGRAMMING IN THE SIMPLY TYPED λ-CALCULUS

This section explains how to use the simply typed λ-calculus to encode compound data structures and proofs in intuitionistic logic. We will use the interpreter as a typed language and, at the same time, as a proof assistant for the intuitionistic propositional logic.

This presentation of simply typed structures follows the Mikrokosmos tutorial and the previous sections on simply typed λ-calculus. All the code on this section is valid Mikrokosmos code.

10.1 FUNCTION TYPES AND TYPEABLE TERMS

Types can be activated with the commmand `:types on`. If types are activated, the interpreter will infer the principal type of every term before its evaluation. The type will then be displayed after the result of the computation.

Example 5 (Typed terms on Mikrokosmos). The following are examples of already defined terms on lambda calculus and their corresponding types. It is important to notice how our previously defined booleans have two different types; while our natural numbers will have all the same type except from zero, whose type is a generalization on the type of the natural numbers.

```
id    --- [1]: λa.a ⇒ id, I, ifelse :: A → A
true  --- [2]: λa.λb.a ⇒ K, true :: A → B → A
false --- [3]: λa.λb.b ⇒ nil, 0, false :: A → B → B
0     --- [4]: λa.λb.b ⇒ nil, 0, false :: A → B → B
1     --- [5]: λa.λb.(a b) ⇒ 1 :: (A → B) → A → B
2     --- [6]: λa.λb.(a (a b)) ⇒ 2 :: (A → A) → A → A
S     --- [7]: λa.λb.λc.((a c) (b c)) ⇒ S :: (A → B → C) → (A → B) → A → C
K     --- [8]: λa.λb.a ⇒ K, true :: A → B → A
```

If a term is found to be non-typeable, Mikrokosmos will output an error message signaling the fact. In this way, the evaluation of λ-terms that could potentially not terminate is prevented. Only typed λ-terms will be evaluated while the option `:types` is on; this ensures the termination of every computation on typed terms.

Example 6 (Non-typeable terms on Mikrokosmos). Fixed point operators are a common example of non typeable terms. Its evaluation on untyped λ-calculus would not terminate; and the type inference algorithm fails on them.

```
fix
--- Error: non typeable expression
fix (\f.\n.iszero n 1 (plus (f (pred n)) (f (pred (pred n))))) 3
--- Error: non typeable expression
```

Note that the evaluation of compound λ-expressions where the fixpoint operators appear applied to other terms can terminate, but the terms are still non typeable.

10.2 PRODUCT, UNION, UNIT AND VOID TYPES

Until now, we have only used the function type. That is to say that we are working on the implicational fragment of the simply-typed lambda calculus we described on the first [BROKEN LINK: *Typing rules for the simply typed \lambda-calculus]. We are now going to extend our type system in the same sense we extended that simply-typed lambda calculus. The following types are added to the system

Type	Name	Description
$\to$	Function type	Functions from a type to another
$\times$	Product type	Cartesian product of types
$+$	Union type	Disjoint union of types
$\top$	Unit type	A type with exactly one element
$\bot$	Void type	A type with no elements

And the following typed constructors are added to the language,

Constructor	Type	Description
`(-,-)`	`A → B → A × B`	Pair of elements
`fst`	`(A × B) → A`	First projection
`snd`	`(A × B) → B`	Second projection
`inl`	`A → A + B`	First inclusion
`inr`	`B → A + B`	Second inclusion
`caseof`	`(A + B) → (A → C) → (B → C) → C`	Case analysis of an union
`unit`	`⊤`	Unital element
`abort`	`⊥ → A`	Empty function
`absurd`	`⊥ → ⊥`	Particular empty function

which correspond to the constructors we described on previous sections. The only new term is the `absurd` function, which is only a particular case of `abort`, useful when we want to make explicit that we are deriving an instance of the empty type. This addition will only make the logical interpretation on the following sections clearer.

Example 7 (Extended STLC on Mikrokosmos). The following are examples of typed terms and functions on Mikrokosmos using the extended typed constructors. The following terms are presented

- a function swapping pairs, as an example of pair types;
- two-case analysis of a number, deciding whether to multiply it by two or to compute its predecessor;
- difference between `abort` and `absurd`;
- example term containing the unit type.

```
:load types
swap = \m.(snd m,fst m)
swap
--- [1]: λa.((SND a),(FST a)) ⇒ swap :: (A × B) → B × A
caseof (inl 1) pred (mult 2)
caseof (inr 1) pred (mult 2)
--- [2]: λa.λb.b ⇒ nil, 0, false :: A → B → B
--- [3]: λa.λb.(a (a b)) ⇒ 2 :: (A → A) → A → A
\x.((abort x),(absurd x))
--- [4]: λa.((ABORT a),(ABSURD a)) :: ⊥ → A × ⊥
```

Now it is possible to define a new encoding of the booleans with an uniform type. The type `⊤ + ⊤` has two inhabitants, `inl ⊤` and `inr ⊤`; and they can be used by case analysis.

```
btrue = inl unit
bfalse = inr unit
bnot = \a.caseof a (\a.bfalse) (\a.btrue)
bnot btrue
--- [1]: (INR UNIT) ⇒ bfalse :: A + ⊤
bnot bfalse
--- [2]: (INL UNIT) ⇒ btrue :: ⊤ + A
```

With these extended types, Mikrokosmos can be used as a proof checker on first-order intuitionistic logic by virtue of the Curry-Howard correspondence.

10.3 a proof in intuitionistic logic

Under the logical interpretation of Mikrokosmos, we can transcribe proofs in intuitionistic logic to λ-terms and check them on the interpreter. The translation between logical propositions and types is straightforward, except for the **negation** of a proposition $\neg A$, that must be written as $(A \to \bot)$, a function to the empty type.

Theorem 8. *In intuitionistic logic, the double negation of the Law of Excluded Middle holds for every proposition, that is,*

$$\forall A\colon \neg\neg(A \vee \neg A)$$

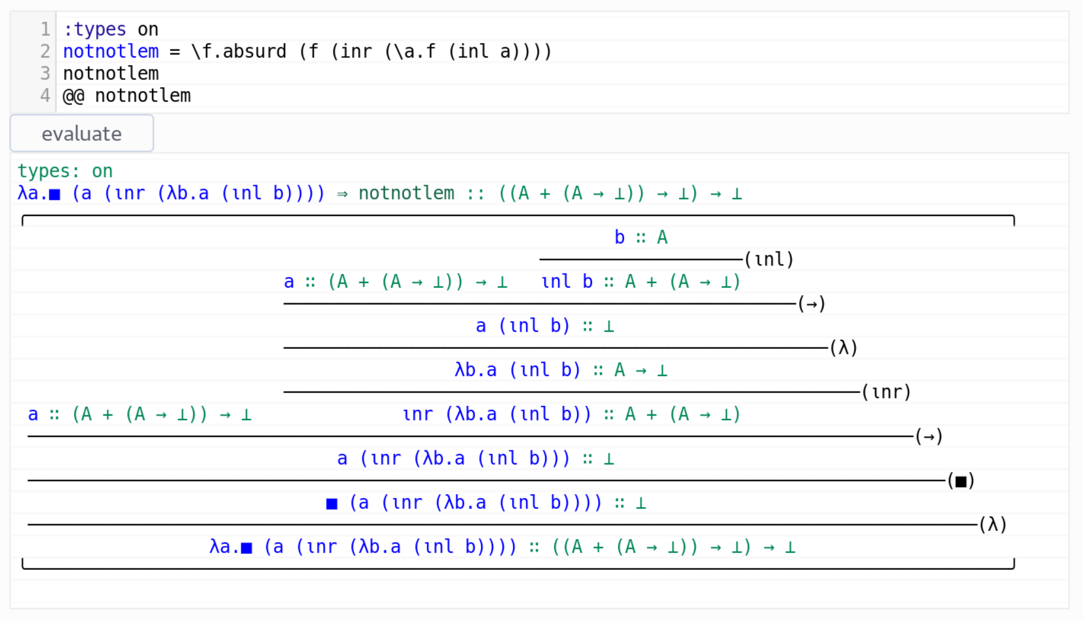
Proof. Suppose $\neg(A \vee \neg A)$. We are going to prove first that, under this specific assumption, $\neg A$ holds. If A were true, $A \vee \neg A$ would be true and we would arrive to a contradition, so $\neg A$. But then, if we have $\neg A$ we also have $A \vee \neg A$ and we arrive to a contradiction with the assumption. We should conclude that $\neg\neg(A \vee \neg A)$. □

Note that this is, in fact, an intuitionistic proof. Although it seems to use the intuitionistically forbidden technique of proving by contradiction, it is actually only proving a negation. There is a difference between assuming A to prove $\neg A$ and assuming $\neg A$ to prove A: the first one is simply a proof of a negation, the second one uses implicitly the law of excluded middle.

This can be translated to the Mikrokosmos implementation of simply typed λ-calculus as the term

```
notnotlem = \f.absurd (f (inr (\a.f (inl a))))
notnotlem
--- [1]: λa.(ABSURD (a (INR λb.(a (INL b))))) :: ((A + (A → ⊥)) → ⊥) → ⊥
```

whose type is precisely $((\mathtt{A} + (\mathtt{A} \to \bot)) \to \bot) \to \bot$. The derivation tree can be seen directly on the interpreter as Figure 1 shows.



```
:types on
notnotlem = \f.absurd (f (inr (\a.f (inl a))))
notnotlem
@@ notnotlem
```

```
types: on
λa.■ (a (ιnr (λb.a (ιnl b)))) ⇒ notnotlem :: ((A + (A → ⊥)) → ⊥) → ⊥
┌───────────────────────────────────────────────────────────────────────────────────────────┐
                                                        b :: A
                                                 ───────────────────(ιnl)
                         a :: (A + (A → ⊥)) → ⊥  ιnl b :: A + (A → ⊥)
                         ────────────────────────────────────────────────(→)
                                           a (ιnl b) :: ⊥
                         ───────────────────────────────────────────────────(λ)
                                         λb.a (ιnl b) :: A → ⊥
                         ──────────────────────────────────────────────────────(ιnr)
 a :: (A + (A → ⊥)) → ⊥             ιnr (λb.a (ιnl b)) :: A + (A → ⊥)
 ───────────────────────────────────────────────────────────────────────────────────(→)
                              a (ιnr (λb.a (ιnl b))) :: ⊥
 ──────────────────────────────────────────────────────────────────────────────────────(■)
                             ■ (a (ιnr (λb.a (ιnl b)))) :: ⊥
 ─────────────────────────────────────────────────────────────────────────────────────────(λ)
                  λa.■ (a (ιnr (λb.a (ιnl b)))) :: ((A + (A → ⊥)) → ⊥) → ⊥
└───────────────────────────────────────────────────────────────────────────────────────────┘
```

Figure 5: Proof of the double negation of LEM.

Part III

CATEGORY THEORY

11

CATEGORIES

We will think of a category as the algebraic structure that captures the notion of composition. A category will be built from some sort of objects linked by composable arrows; to which associativity and identity laws will apply.

Thus, a category has to rely in some notion of ***collection***. When interpreted inside set-theory, it is common to use this term to denote some unspecified formal notion of compilation of entities that could be given by sets or proper classes. We will want to define categories whose objects are all the possible sets and we will need the objects to form a proper class in order to avoid inconsistent results such as the Russell's paradox. This is why we will consider, from this approach, a particular class of categories of small set-theoretical size to be specially well-behaved.

Definition 29 (Small and locally small categories)**.** A category will be said to be **small** if the collection of its objects can be given by a set (instead of a proper class). It will be said to be **locally small** if the collection of arrows between any two objects can be given by a set.

A different approach, however, would be to simply take the *objects* and the *arrows* as fundamental concepts of our theory. These foundational concerns will not cause any explicit problem in this presentation of category theory, so we will keep them deliberately open to both interpretations.

11.1 DEFINITION OF CATEGORY

Definition 30 (Category)**.** A **category** $\mathcal{C}$, as defined in [Lan78], is given by

- $\mathcal{C}_0$ (sometimes denoted $\mathrm{obj}(\mathcal{C})$ or simply $\mathcal{C}$), a *collection* whose elements are called **objects**, and
- $\mathcal{C}_1$, a *collection* whose elements are called **morphisms**.

Every morphism $f \in \mathcal{C}_1$ is assigned two objects: a **domain**, written as $\mathrm{dom}(f) \in \mathcal{C}_0$, and a **codomain**, written as $\mathrm{cod}(f) \in \mathcal{C}_0$; a common notation for such morphism is

$$f\colon \mathrm{dom}(f) \to \mathrm{cod}(f).$$

Given two morphisms $f\colon A \to B$ and $g\colon B \to C$, there exists a **composition morphism**, written as $g \circ f\colon A \to C$; or simply by yuxtaposition, as gf. Morphism composition is a binary associative operation with an identity element $\mathrm{id}_A\colon A \to A$ for every object A, that is,

$$h \circ (g \circ f) = (h \circ g) \circ f \quad \text{and} \quad f \circ \mathrm{id}_A = f = \mathrm{id}_B \circ f,$$

for any f, g, h, composable morphisms.

Definition 31 (Hom-sets)**.** The **hom-set** of two objects A, B on a category is the collection of morphisms between them. It is written as $\hom(A, B)$. The set of **endomorphisms** of an object A is defined as $\mathrm{end}(A) = \hom(A, A)$.

We can use a subscript, as in $\hom_{\mathcal{C}}(A, B)$ to explicitly specify the category we are working in when necessary.

11.2 MORPHISMS

Objects in category theory are an atomic concept and can be only studied by their morphisms; that is, by how they are related to all the objects of the category. Thus, the essence of a category is given not by its objects, but by the morphisms between them and how composition is defined.

It is so much so, that we will consider two objects essentially equivalent (and we will call them ***isomorphic***) whenever they relate to other objects in the exact same way; that is, whenever an invertible morphism between them exists. This will constitute an equivalence relation on the category.

In a certain sense, morphisms are an abstraction of the notion of the structure-preserving homomorphisms that are defined between algebraic structures. From this perspective, ***monomorphisms*** and ***epimorphisms*** can be thought as abstractions of the usual injective and surjective homomorphisms. We will see, however, how some properties that we take for granted, such as "isomorphism" meaning exactly the same as "both injective and surjective", are not true in general.

Definition 32 (Isomorphisms)**.** A morphism $f : A \to B$ is an **isomorphism** if there exist a morphism $f^{-1} : B \to A$ such that

- $f^{-1} \circ f = \mathrm{id}_A$,
- $f \circ f^{-1} = \mathrm{id}_B$.

This morphism is called an *inverse morphism*.

We call **automorphisms** to the morphisms which are both endomorphisms and isomorphisms.

Proposition 2 (Unicity of inverses)**.** *If the inverse of a morphism exists, it is unique. In fact, if a morphism has a left-side inverse and a right-side inverse, they are both-sided inverses and they are equal.*

Proof. Given $f : A \to B$ with inverses $g_1, g_2 : B \to A$; we have that

$$g_1 = g_1 \circ \mathrm{id}_A = g_1 \circ (f \circ g_2) = (g_1 \circ f) \circ g_2 = \mathrm{id} \circ g_2 = g_2.$$

We have used associativity of composition, neutrality of the identity and the fact that g_1 is a left-side inverse and g_2 is a right-side inverse. □

Definition 33. Two objects are **isomorphic** if an isomorphism between them exists. We write $A \cong B$ when A and B are isomorphic.

Proposition 3 (Isomorphy is an equivalence relation)**.** *The relation of being isomorphic is an equivalence relation. In particular,*

- *the identity,* $\mathrm{id} = \mathrm{id}^{-1}$*;*
- *the inverse of an isomorphism,* $(f^{-1})^{-1} = f$*;*
- *and the composition of isomorphisms,* $(f \circ g)^{-1} = g^{-1} \circ f^{-1}$*;*

are all isomorphisms.

Proof. We can check that those are in fact inverses. From their existance follows

- reflexivity, $A \cong A$;
- symmetry, $A \cong B$ implies $B \cong A$;
- transitivity, $A \cong B$ and $B \cong C$ imply $A \cong C$.

□

Definition 34 (Monomorphisms and epimorphisms)**.** A **monomorphism** is a left-cancellable morphism, that is, $f : A \to B$ is a monomorphism if, for every $g, h : B \to A$,

$$f \circ g = f \circ h \implies g = h.$$

An **epimorphism** is a right-cancellable morphism, that is, $f : A \to B$ is an epimorphism if, for every $g, h : B \to A$,

$$g \circ f = h \circ f \implies g = h.$$

A morphism that is a monomorphism and an epimorphism at the same time is called a **bimorphism**.

Remark 1. A morphism can be a bimorphism without being an isomorphism. We will cover examples of this fact later.

Definition 35 (Retractions and sections)**.** A **retraction** is a left inverse, that is, a morphism that has a right inverse; conversely, a **section** is a right inverse, a morphism that has a left inverse.

By virtue of Proposition 2, a morphism that is both a retraction and a section is an isomorphism. Thus, not every epimorphism is a section and not every monomorphism is a retraction.

11.3 terminal objects, products and coproducts

Products and coproducts are very widespread notions in mathematics. Whenever a new structure is defined, it is common to wonder what the product or sum of two of these structures would be. Examples of products are the cartesian product of sets, the product topology or the product of abelian groups; examples of coproducts are the disjoint union of sets, topological sum or the free product of groups.

We will abstract categorically these notions in terms of *universal properties*. This viewpoint, however, is an important shift with respect to how these properties are usually defined. We will not define the product of two objects in terms of its internal structure (categorically, objects are atomic and do not have any); but in terms of all the other objects, that is, in terms of the complete structure of the category. This turns inside-out the focus of definitions. Moreover, objects defined in terms of universal properties are usually not uniquely determined, but only determined up to isomorphism. This reinforces our previous idea of considering two isomorphic objects in a category as *essentialy* the same object.

Initial and terminal objects will be a first example of this viewpoint based on universal properties.

Definition 36 (Initial object)**.** An object I is an **initial object** if every object is the domain of exactly one morphism from it. That is, for every object A exists an unique morphism $i_A \colon I \to A$.

Definition 37 (Terminal object)**.** An object T is a **terminal object** (also called *final object*) if every object is the codomain of exactly one morphism to it. That is, for every object A exists an unique $t_A \colon A \to T$.

Definition 38 (Zero object)**.** A **zero object** is an object which is both initial and terminal at the same time.

Proposition 4 (Initial and final objects are essentially unique)**.** *Initial and final objects in a category are essentially unique; that is, any two initial objects are isomorphic and any two final objects are isomorphic.*

Proof. If A, B were initial objects, by definition, there would be only one morphism $f : A \to B$ and only one morphism $g : B \to A$. Moreover, there would be only an endomorphism in $\mathrm{End}(A)$ and $\mathrm{End}(B)$ which should be the identity. That implies,

- $f \circ g = \mathrm{id}$,
- $g \circ f = \mathrm{id}$.

As a consequence, $A \cong B$. A similar proof can be written for the terminal object. □

Note, however, that these objects may not exist in any given category.

Definition 39 (Product object)**.** An object C is the **product** of two objects A, B on a category if there are two morphisms

$$A \xleftarrow{\pi_A} C \xrightarrow{\pi_B} B$$

such that, for any other object D with two morphisms $f_1 : D \to A$ and $f_2 : D \to B$, an unique morphism $h : D \to C$, such that $f_1 = \pi_A \circ h$ and $f_2 = \pi_B \circ h$. Diagramatically,

$$\begin{tikzcd}
& D \arrow[d, dashed, "\exists! h"] \arrow[ddl, bend right, "f_1"'] \arrow[ddr, bend left, "f_2"] & \\
& C \arrow[dl, "\pi_A"'] \arrow[dr, "\pi_B"] & \\
A & & B
\end{tikzcd}$$

Note that the product of two objects does not have to exist on a category; but when it exists, it is essentially unique. In fact, we will be able later to construct a category in which the product object is the final object of the category and Proposition 4 can be applied. We will write *the* product object of A, B as $A \times B$.

Definition 40 (Coproduct object)**.** An object C is the **coproduct** of two objects A, B on a category if there are two morphisms

$$A \xrightarrow{i_A} C \xleftarrow{i_B} B$$

such that, for any other object D with two morphisms $f_1 : D \to A$ and $f_2 : D \to B$, an unique morphism $h : D \to C$, such that $f_1 = i_A \circ h$ and $f_2 = i_B \circ h$. Diagramatically,

$$\begin{tikzcd}
& D & \\
& C \arrow[u, dashed, "\exists! h"] & \\
A \arrow[uur, bend left, "f_1"] \arrow[ur, "i_A"] & & B \arrow[uul, bend right, "f_2"'] \arrow[ul, "i_B"']
\end{tikzcd} .$$

The same discussion we had earlier for the product can be rewritten here for the coproduct only reversing the direction of the arrows. We will write *the* coproduct of A, B as $A \amalg B$. As we will see later, the notion of a coproduct is dual to the notion of product; and the same proofs can be applied on both cases, only by reversing the arrows.

11.4 examples of categories

Example 8 (Discrete categories)*.* A category is **discrete** if it has no other morphisms than the identities. A discrete category is uniquely defined by the class of its objects and every class of objects defines a discrete category. Thus, discrete categories are classes or sets, without any additional categorical structure.

Example 9 (Monoids, groups). A single-object category is a **monoid**. A monoid in which every morphism is an isomorphism is a **group**.

This definition is equivalent to the usual definition of monoid if we take the morphisms as elements of the monoid and composition of morphisms as the monoid operation. Groupoids are also a particular case of categories.

Example 10 (Groupoids). A category in which every morphism is an isomorphism is a **groupoid**.

Example 11 (Partially ordered sets). Every partial ordering defines a category in which the elements are the objects and an only morphism between two objects $\rho_{a,b} : a \to b$ exists

In particular, every ordinal can be seen as a partially ordered set and defines a category.

For example, if we take the finite ordinal $[n] = (0 < \cdots < n)$, it could be interpreted as the category given by the following diagram

$$\begin{tikzcd}
0 \arrow[loop above] \arrow[r] \arrow[rr, bend right] \arrow[rrrr, bend right] & 1 \arrow[loop above] \arrow[r] \arrow[rrr, bend right] & 2 \arrow[loop above] \arrow[r] \arrow[rr, bend right] & \dots \arrow[r] & n \arrow[loop above]
\end{tikzcd}$$

in which every object p has an identity arrow and a unique arrow to every q such that $p \leq q$. Note how the composition of arrows can be only defined in a single way.

In a partially ordered set, the product of two objects would be its join, the coproduct would be its meet and the initial and terminal objects would be the greatest and the least element, respectively.

Example 12 (The category of sets). The category Set is defined as the category with all the possible sets as objects and functions between them as morphisms. It is trivial to check associativity of composition and the existence of the identity function for any set.

In this category, the product is given by the usual cartesian product

$$A \times B = \{(a, b) \mid a \in A,\ b \in B\},$$

with the projections $\pi_A(a, b) = a$ and $\pi_B(a, b) = b$. We can easily check that, if we have $f : C \to A$ and $g : C \to B$, there is a unique function given by $h(c) = (f(c), g(c))$ such that $\pi_A \circ h = f$ and $\pi_B \circ h = g$.

The initial object in Set is given by the empty set $\varnothing$: given any set A, the only function of the form $f : \varnothing \to A$ is the empty one. The final object, however, is only defined up to isomorphism: given any set with a single object $\{*\}$, there exists a unique function of the form $f : A \to \varnothing$ for any set A; namely, the one defined as $\forall a \in A : f(a) = *$. Every two sets with exactly one object are terminal objects and they are trivially isomorphic.

Similarly, the coproduct is defined only to isomorphism. The coproduct of two sets A, B is given by its disjoint union $A \sqcup B$; but this union can be defined in many different (but equivalent) ways. For instance, we can add a label to the elements of each sets before joining them in order to ensure that this will be a disjoint union; that is,

$$A \sqcup B = \{(a, 0) \mid a \in A\} \cup \{(b, 1) \mid b \in B\}$$

with the inclusions $i_A(a) = (a, 0)$ and $i_B(b) = (b, 1)$, is a possible coproduct. Given any two functions $f : A \to C$ and $g : A \to C$, there exists a unique function $h : A \sqcup B \to C$, given by

$$h(x, n) = \begin{cases} f(x) & \text{if } n = 0, \\ g(x) & \text{if } n = 1, \end{cases}$$

such that $f = h \circ i_A$ and $g = h \circ i_B$.

The category of sets is a very special category, whose properties we will study in detail later.

Example 13 (The category of groups). The category Grp is defined as the category with groups as objects and group homomorphisms between them as morphisms.

Example 14 (The category of R-modules). The category R-Mod is defined as the category with R-modules as objects and module homomorphisms between them as morphisms. We know that the composition of module homomorphisms and the identity are also module homomorphisms.

In particular, abelian groups form a category as $\mathbb{Z}$-modules.

Example 15 (The category of topological spaces). The category Top is defined as the category with topological spaces as objects and continuous functions between them as morphisms.

12

FUNCTORS AND NATURAL TRANSFORMATIONS

> "Category" has been defined in order to define "functor" and "functor" has been defined in order to define "natural transformation".
>
> – **Saunders MacLane**, *Categories for the working mathematician*, [EM42].

Functors and natural transformations were defined for the first time by Eilenberg and MacLane in [EM42] while studying Čech cohomology. While initially they were devised mainly as a language for studying homology, they have proven its foundational value with the passage of time. The notion of naturality will be a key element of our presentation of algebraic theories and categorical logic.

12.1 FUNCTORS

A ***functor*** will be interpreted as a homomorphism of categories preserving their structure. As we discussed in the previous section, the structure of a category is given by the composition of morphisms.

Definition 41 (Functor)**.** Given two categories $\mathcal{C}$ and $\mathcal{D}$, a **functor** between them, $F : \mathcal{C} \to \mathcal{D}$, is given by

- an **object function**, $F : \mathrm{obj}(\mathcal{C}) \to \mathrm{obj}(\mathcal{D})$;
- and an **arrow function**, $F : \hom(A, B) \to \hom(FA, FB)$ for any two objects A, B of the category;

such that

- $F(\mathrm{id}_A) = \mathrm{id}_{FA}$, identities are preserved; and
- $F(f \circ g) = Ff \circ Fg$, the functor respects composition.

Functors can be composed as we did with morphisms. In fact, a category of categories can be defined, having functors as morphisms. In order to avoid paradoxes, we will only define the category of all small categories as a non-small category so it will not contain itself.

Definition 42 (The category of categories)**.** The category Cat is defined as the category of (small) categories as objects and functors as morphisms.

- Given two functors $F\colon \mathcal{C} \to \mathcal{B}$ and $G\colon \mathcal{B} \to \mathcal{A}$, their composite functor $G \circ F : \mathcal{C} \to \mathcal{A}$ is given by the composition of the object and arrow functions of the functors. This composition is trivially associative.
- The identity functor on a category $I_{\mathcal{C}}\colon \mathcal{C} \to \mathcal{C}$ is given by identity object and arrow functions. It is trivially neutral with respect to composition.

Definition 43 (Full functor)**.** A functor F is **full** if the arrow map between any pair of objects is surjective. That is, if every $g : FA \to FB$ is of the form Ff for some morphism $f\colon A \to B$.

Definition 44 (Faithful functor)**.** A functor F is **faithful** if the arrow map between any pair of objects is injective. That is, if, for every two arrows $f_1, f_2 : A \to B$, $Ff_1 = Ff_2$ implies $f_1 = f_2$.

It is easy to notice that the composition of faithful (respectively, full) functors is again a faithful functor (respectively, full).

Note that a faithful functor needs not to be injective on objects nor on morphisms. In particular, if A, A', B, B' are four different objects, it could be the case that $FA = FA'$ and $FB = FB'$; and, if $f : A \to B$ and $f' : A' \to B'$ were two morphisms, it could be the case that $Ff = Ff'$.

Definition 45 (Isomorphism of categories)**.** An **isomorphism of categories** is a functor T whose object and arrow functions are bijections. Equivalently, it is a functor T such that there exists an *inverse* functor S such that $T \circ S$ and $S \circ T$ are identity functors.

However, the notion of isomorphism of categories may be too strict. Sometimes, it will suffice if the two compositions $T \circ S$ and $S \circ T$ are not exactly the identity functor, but isomorphic in some sense to it. We will develop these weaker notions in the next section.

12.2 NATURAL TRANSFORMATIONS

Definition 46 (Natural transformation)**.** A **natural transformation** between two functors with the same domain and codomain, $\alpha\colon F \stackrel{\cdot}{\to} G$, is a family of morphisms parameterized by the objects of the domain category, $\alpha_C\colon FC \to GC$ such that the following diagram commutes

$$\begin{tikzcd}
C \arrow[d, "f"] & & SC \arrow[r, "\tau_C"] \arrow[d, "Sf"'] & TC \arrow[d, "Tf"] \\
C' & & SC' \arrow[r, "\tau_{C'}"] & TC'
\end{tikzcd}$$

for every arrow $f : C \to C'$.

Sometimes, it is also said that the family of morphisms τ is *natural* in its argument. This naturality property is what allows us to "translate" a commutative diagram from a functor to another.

$$\begin{tikzcd}
A \arrow[dr, "f"] \arrow[dd, "h"'] & \\
& B \arrow[dl, "g"] \\
C &
\end{tikzcd}
\qquad
\begin{tikzcd}
FA \arrow[rr, "\tau A"] \arrow[dr, "Ff"] \arrow[dd, "Fh"'] & & GA \arrow[dr, "Gf"] \arrow[dd] & \\
& FB \arrow[rr, "\tau B"] \arrow[dl, "Fg"] & & GB \arrow[dl, "Gg"] \\
FC \arrow[rr, "\tau C"] & & GC &
\end{tikzcd}$$

Definition 47 (Natural isomorphism)**.** A **natural isomorphism** is a natural transformation in which every component, every morphism of the parameterized family, is invertible.

The inverses of a natural transformation form another natural transformation, whose naturality follows from the naturality of the original transformation. We say that two functors T, S are **naturally isomorphic**, and we write this as $T \cong S$, if there is a natural isomorphism between them. The notion of a natural isomorphism between functors allows us to weaken the condition of strict equality that we imposed when talking about isomorphisms of categories. The generally more useful notion of *equivalence of categories* only needs the composition of the two functors to be naturally isomorphic to the identity.

Definition 48 (Equivalence of categories)**.** An **equivalence of categories** is given by two functors T and S such that its two compositions are naturally isomorphic to the identity functor, $T \circ S \cong \mathrm{id}$ and $S \circ T \cong \mathrm{id}$.

12.3 composition of natural transformations

There is an obvious way in which two natural transformations $\sigma : R \to S$ and $\tau : S \to T$ can be composed into a new natural transformation $R \to T$; this will be used later to define categories whose objects are functors and whose morphisms are natural transformations. But there is also a different notion of composition of natural transformations, which applies two natural transformations, in parallel, to the composition of two functors.

That is, if we draw a natural transformation between two functors as a double arrow

$$\begin{tikzcd}
A \arrow[r, bend left=50, "S"{name=U}] \arrow[r, bend right=50, "T"'{name=D}] & B
\arrow[Rightarrow, from=U, to=D, "\tau"]
\end{tikzcd}$$

- we have a *vertical* composition of natural transformations, which, diagramatically, composes the two natural transformations of the left diagram into a transformation like in the right one

$$\begin{tikzcd}
A \arrow[rr, bend left=60, "R", ""{name=U, below}] \arrow[rr, "T"' {name=M}] \arrow[rr, bend right=60, "S"', ""{name=D, above}] & & B
\arrow[Rightarrow, from=U, to=M, "\tau"] \arrow[Rightarrow, from=M, to=D, "\sigma"]
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[rr, bend left=50, "R", ""{name=U, below}] \arrow[rr, bend right=50, "T"', ""{name=D, above}] & & B
\arrow[Rightarrow, from=U, to=D, "\tau\cdot\sigma"]
\end{tikzcd}$$

- and we have a *horizontal* composition of natural transformations, which composes the two natural transformations of the first diagram into the second one

$$\begin{tikzcd}
A \arrow[rr, bend left=50, "S", ""{name=U1, below}] \arrow[rr, bend right=50, "T"', ""{name=D1, above}] & & B \arrow[rr, bend left=50, "S'", ""{name=U2, below}] \arrow[rr, bend right=50, "T'"', ""{name=D2, above}] & & C
\arrow[Rightarrow, from=U1, to=D1, "\tau"] \arrow[Rightarrow, from=U2, to=D2, "\sigma"]
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[rr, bend left=50, "S\circ S'", ""{name=U, below}] \arrow[rr, bend right=50, "T\circ T'"', ""{name=D, above}] & & C
\arrow[Rightarrow, from=U, to=D, "\tau\circ\sigma"]
\end{tikzcd}$$

Definition 49 (Vertical composition of natural transformations)**.** The **vertical composition** of two natural transformations $\tau : S \to T$ and $\sigma : R \to S$, denoted by $\tau \cdot \sigma$ is the family of morphisms defined by the objectwise composition of the components of the two natural transformations, that is

$$\begin{tikzcd}
Rc \arrow[r, "Rf"] \arrow[d, "\sigma_c"] \arrow[dd, bend right=80, "(\tau\circ\sigma)_c"'] & Rc' \arrow[d, "\sigma_{c'}"] \arrow[dd, bend left=80, "(\tau\circ\sigma)_{c'}"] \\
Sc \arrow[r, "Sf"] \arrow[d, "\tau_c"] & Sc' \arrow[d, "\tau_{c'}"] \\
Tc \arrow[r, "Tf"] & Tc'
\end{tikzcd}$$

Proposition 5 (Vertical composition is a natural transformation)**.** *The vertical composition of two natural transformations is in fact a natural transformation.*

Proof. Naturality of the composition follows from the naturality of its two factors. In other words, the commutativity of the external square on the above diagram follows from the commutativity of the two internal squares. □

Definition 50 (Horizontal composition of natural transformations)**.** The **horizontal composition** of two natural transformations $\tau \colon S \to T$ and $\tau' \colon S' \to T'$, with domains and codomains as in the following diagram

$$\begin{tikzcd}
A \arrow[rr, bend left=50, "S", ""{name=U1, below}] \arrow[rr, bend right=50, "T"', ""{name=D1, above}] & & B \arrow[rr, bend left=50, "S'", ""{name=U2, below}] \arrow[rr, bend right=50, "T'"', ""{name=D2, above}] & & C
\arrow[Rightarrow, from=U1, to=D1, "\tau"] \arrow[Rightarrow, from=U2, to=D2, "\sigma"]
\end{tikzcd}$$

is denoted by $\tau' \circ \tau \colon S'S \to T'T$ and is defined as the family of morphisms given by $\tau' \circ \tau = T'\tau \circ \tau' = \tau' \circ S'\tau$, that is, by the diagonal of the following commutative square

$$\begin{tikzcd}
S'Sc \arrow[r, "\tau'_{Sc}"] \arrow[d, "S'\tau_c"'] \arrow[dr, "(\tau' \circ \tau)_c"] & T'Sc \arrow[d, "T'\tau_c"] \\
S'Tc \arrow[r, "\tau'_{Tc}"] & T'Tc
\end{tikzcd}$$

Proposition 6 (Horizontal composition is a natural transformation)**.** *The horizontal composition of two natural transformations is in fact a natural transformation.*

Proof. It is natural as the following diagram is the composition of two naturality squares

$$\begin{tikzcd}
S'Sc \arrow[r, "S'\tau"] \arrow[d, "S'Sf"] & S'Tc \arrow[r, "\tau'"] \arrow[d, "S'Tf"] & T'Tc \arrow[d, "T'Tf"] \\
S'Sb \arrow[r, "S'\tau"] & S'Tb \arrow[r, "\tau'"] & T'Tb
\end{tikzcd}$$

defined respectively by the naturality of $S'\tau$ and τ'. □

13

CONSTRUCTIONS ON CATEGORIES

13.1 PRODUCT CATEGORIES

Definition 51 (Product category)**.** The **product category** of two categories $\mathcal{C}$ and $\mathcal{D}$, denoted by $\mathcal{C} \times \mathcal{D}$ is a category having

- pairs $\langle c, d\rangle$ as objects, where $c \in \mathcal{C}$ and $d \in \mathcal{D}$;
- and pairs $\langle f, g\rangle : \langle c, d\rangle \to \langle c', d'\rangle$ as morphisms, where $f : c \to c'$ and $g : d \to d'$ are morphisms in their respective categories.

The identity morphism of any object $\langle c, d\rangle$ is $\langle \mathrm{id}_c, \mathrm{id}_d\rangle$, and composition is defined componentwise as

$$\langle f', g'\rangle \circ \langle f, g\rangle = \langle f' \circ f, g' \circ g\rangle.$$

We also define **projection functors** $P : \mathcal{C} \times \mathcal{D} \to \mathcal{C}$ and $Q : \mathcal{C} \times \mathcal{D} \to \mathcal{D}$ on arrows as $P\langle f, g\rangle = f$ and $Q\langle f, g\rangle = g$. Note that this definition of product, using these projections, would be the product of two categories on a category of categories with functors as morphisms.

Definition 52 (Product of functors)**.** The **product functor** of two functors $F : \mathcal{C} \to \mathcal{C}'$ and $G : \mathcal{D} \to \mathcal{D}'$ is a functor $F \times G : \mathcal{C} \times \mathcal{D} \to \mathcal{C}' \times \mathcal{D}'$ which can be defined

- on objects as $(F \times G)\langle c, d\rangle = \langle Fc, Gd\rangle$;
- and on arrows as $(F \times G)\langle f, g\rangle = \langle Ff, Gg\rangle$.

It can be seen as the unique functor making the following diagram commute

$$\begin{tikzcd}
\mathcal{C} \arrow[d, "F"] & \mathcal{C} \times \mathcal{D} \arrow[l, "P"'] \arrow[r, "Q"] \arrow[d, dashed, "F \times G"] & \mathcal{D} \arrow[d, "G"] \\
\mathcal{C}' & \mathcal{C}' \times \mathcal{D}' \arrow[l, "P'"] \arrow[r, "Q'"'] & \mathcal{D}'
\end{tikzcd}$$

In this sense, the $\times$ operation is itself a functor acting on objects and morphisms of the `Cat` category of all categories. A **bifunctor** is a functor from a product category;

and it also can be seen as a functor on two variables. As we will show in the following proposition, it is completely determined by the two families of functors that we obtain when we fix any of the elements.

Proposition 7 (Conditions for the existence of bifunctors). *Let $\mathcal{B}, \mathcal{C}, \mathcal{D}$ categories with two families of functors*

$$\{L_c \colon \mathcal{B} \to \mathcal{D}\}_{c \in \mathcal{C}} \quad \textit{and} \quad \{M_b \colon \mathcal{C} \to \mathcal{D}\}_{b \in \mathcal{B}},$$

such that $M_b(c) = L_c(b)$ for all b, c. A bifunctor $S \colon \mathcal{B} \times \mathcal{C} \to \mathcal{D}$ such that $S(-, c) = L_c$ and $S(b, -) = M_b$ for all b, c exists if and only if for every $f \colon b \to b'$ and $g \colon c \to c'$,

$$M_{b'}g \circ L_c f = L_{c'} f \circ M_b g.$$

Proof. If the equality holds, the bifunctor can be defined as $S(b, c) = M_b(c) = L_c(b)$ in objects and as $S(f, g) = M_{b'}g \circ L_c f = L_{c'} f \circ M_b g$ on morphisms. This bifunctor preserves identities, as

$$S(\mathrm{id}_b, \mathrm{id}_c) = M_b(\mathrm{id}_c) \circ L_c(\mathrm{id}_b) = \mathrm{id}_{M_b(c)} \circ \mathrm{id}_{L_c(b)} = \mathrm{id},$$

and it preserves composition, as

$$S(f', g') \circ S(f, g) = Mg' \circ Lf' \circ Mg \circ Lf = Mg' \circ Mg \circ Lf' \circ Lf = S(f' \circ f, g' \circ g)$$

for any composable morphisms f, f', g, g'. On the other hand, if a bifunctor exists,

$$\begin{aligned} M_{b'}(g) \circ L_c(f) &= S(\mathrm{id}_{b'}, g) \circ S(f, \mathrm{id}_c) = S(\mathrm{id}_{b'} \circ f, g \circ \mathrm{id}_c) \\ &= S(f \circ \mathrm{id}_b, \mathrm{id}_{c'} \circ g) = S(f, \mathrm{id}_{c'}) \circ S(\mathrm{id}_b, g) \\ &= L_{c'}(f) \circ M_b(f). \end{aligned}$$

□

Proposition 8 (Naturality for bifunctors). *Given S, S' bifunctors, $\alpha_{b,c} \colon S(b, c) \to S'(b, c)$ is a natural transformation if and only if $\alpha(b, c)$ is natural in b for each c and natural in c for each b.*

Proof. If α is natural, in particular, we can use the identities to prove that it must be natural in its two components

$$\begin{tikzcd} S(b,c) \arrow[r, "\alpha_{b,c}"] \arrow[d, "S\langle f{,}\mathrm{id}_c\rangle"'] & S'(b,c) \arrow[d, "S'\langle f{,}\mathrm{id}_c\rangle"] \\ S(b',c) \arrow[r, "\alpha_{b',c}"] & S'(b',c) \end{tikzcd} \qquad \begin{tikzcd} S(b,c) \arrow[r, "\alpha_{b,c}"] \arrow[d, "S\langle \mathrm{id}_b{,}g\rangle"'] & S'(b,c) \arrow[d, "S'\langle \mathrm{id}_b{,}g\rangle"] \\ S(b,c') \arrow[r, "\alpha_{b,c'}"] & S'(b,c') \end{tikzcd}$$

If both components of α are natural, the naturality of the natural transformation follows from the composition of these two squares

$$\begin{tikzcd}
S(b,c) \arrow[r, "\alpha_{b,c}"] \arrow[d, "S\langle f{,}\mathrm{id}_c\rangle"'] & S'(b,c) \arrow[d, "S'\langle f{,}\mathrm{id}_c\rangle"] \\
S(b',c) \arrow[r, "\alpha_{b',c}"] \arrow[d, "S\langle \mathrm{id}_{b'}{,}g\rangle"'] & S'(b',c) \arrow[d, "S'\langle \mathrm{id}_{b'}{,}g\rangle"] \\
S(b',c') \arrow[r, "\alpha_{b',c'}"] & S'(b',c')
\end{tikzcd}$$

where each square is commutative by the naturality of each component of α. □

13.2 opposite categories and contravariant functors

Definition 53 (Opposite category)**.** The **opposite category** $\mathcal{C}^{op}$ of a category $\mathcal{C}$ is a category with the same objects as $\mathcal{C}$ but with all its arrows reversed. That is, for each morphism $f : A \to B$, there exists a morphism $f^{op} : B \to A$ in $\mathcal{C}^{op}$. Composition is defined as

$$f^{op} \circ g^{op} = (g \circ f)^{op},$$

exactly when the composite $g \circ f$ is defined in $\mathcal{C}$.

Reversing all the arrows is a process that directly translates every property of the category into its ***dual*** property. A morphism f is a monomorphism if and only if f^{op} is an epimorphism; a terminal object in $\mathcal{C}$ is an initial object in $\mathcal{C}^{op}$ and a right inverse becomes a left inverse on the opposite category. This process is also an *involution*, where $(f^{op})^{op}$ can be seen as f and $(\mathcal{C}^{op})^{op}$ is trivially isomorphic to $\mathcal{C}$.

Definition 54 (Contravariant functor)**.** A **contravariant** functor from $\mathcal{C}$ to $\mathcal{D}$ is a functor from the opposite category, that is, $F\colon \mathcal{C}^{op} \to \mathcal{D}$. Non-contravariant functors are often called **covariant** functors, to emphasize the difference.

Example 16 (Hom functors). In a locally small category $\mathcal{C}$, the **Hom-functor** is the bifunctor

$$\mathrm{hom}\colon \mathcal{C}^{op} \times \mathcal{C} \to \mathsf{Set},$$

defined as $\mathrm{hom}(a,b)$ for any two objects $a,b \in \mathcal{C}$. Given $f\colon a \to a'$ and $g\colon b \to b'$, this functor is defined on any $p \in \mathrm{hom}(a,b)$ as

$$\mathrm{hom}(f,g)(p) = f \circ p \circ g \in \mathrm{hom}(a',b').$$

Partial applications of the functors give rise to

- $\hom(a, -)$, a covariant functor for any fixed $a \in \mathcal{C}$. Given $g \colon b \to b'$,

  $$\hom(a, f) \colon \hom(a, b) \to \hom(a, b')$$

  is defined as the postcomposition with g, that we write as $- \circ g$.
- $\hom(-, b)$, a contravariant functor for any fixed $b \in \mathcal{C}$. Given $f \colon a \to a'$,

  $$\hom(f, b) \colon \hom(a', b) \to \hom(a, b)$$

  is defined as the precomposition with f, that we write as $f \circ -$.

This kind of functor, contravariant on the first variable and covariant on the second is usually called a **profunctor**.

13.3 FUNCTOR CATEGORIES

Definition 55 (Functor category)**.** Given two categories $\mathcal{B}, \mathcal{C}$, the **functor category** $\mathcal{B}^{\mathcal{C}}$ has all functors $\mathcal{C} \to \mathcal{B}$ as objects and natural transformations between them as morphisms.

If we consider the category of small categories `Cat`, there is a profunctor $-^{-} \colon \texttt{Cat}^{op} \times \texttt{Cat} \to \texttt{Cat}$ sending any two categories to their functor category.

In [LS09], multiple examples of usual mathematical constructions in terms of functor categories can be found. Graphs, for instance, can be seen as functors; and graphs homomorphisms as the natural transformations between them.

Example 17 (Graphs as functors). We consider the category given by two objects and two non-identity morphisms,

$$\begin{tikzcd} \cdot \arrow[r, bend left] \arrow[r, bend right] & \cdot \end{tikzcd}$$

usually called $\downarrow\downarrow$. To define a functor from this category to Set amounts to choose two sets E, V (not necessarily different) called the set of *edges* and the set of *vertices*; and two functions $s, t \colon E \to V$, called *source* and *target*. That is, our usual definition of directed multigraph,

$$\begin{tikzcd} E \arrow[r, bend left, "s"] \arrow[r, bend right, "t"'] & V \end{tikzcd}$$

can be seen as an object in the category $\mathrm{Set}^{\downarrow\downarrow}$. Note how a natural transformation between two graphs (E, V) and (E', V') is a pair of morphisms $\alpha_E \colon E \to E'$ and $\alpha_V \colon V \to V'$ such that $s \circ \alpha_E = \alpha_V \circ s$ and $t \circ \alpha_E = \alpha_V \circ t$. This provides a convenient notion of graph homomorphism: a pair of morphisms preserving the incidence of edges. We can call `Graph` to this functor category.

Example 18 (Dynamical systems as functors). A set endowed with an endomorphism (S, α) can be regarded as a *dynamical system* in an informal way. Each state of the system is represented by an element of the set and and the transition function is represented by the endomorphism. That is, if we start at an initial state $s \in S$ and the transition function is given by $\alpha \colon S \to S$, the evolution of the system will be given by

$$s,\ \alpha(s),\ \alpha(\alpha(s)),\ \dots$$

and we could say that it evolves discretely over time, being $\alpha^t(s)$ the state of the system at the instant t.

This structure can be described as a functor from the monoid of natural numbers under addition. Note that any functor $D \colon \mathbb{N} \to \mathsf{Set}$ has to choose a set, and an image for the $1 : \mathbb{N} \to \mathbb{N}$, the only generator of the monoid. The image of any natural number n is determined by the image of 1; if $D(1) = \alpha$, it follows that $D(t) = \alpha^t$, where $\alpha^0 = \mathrm{id}$.

Once the structure has been described as a functor, the homomorphisms preserving this kind of structure can be described as natural transformations. A natural transformation between two functors $D, D' \colon \mathbb{N} \to \mathsf{Set}$ describing two dynamic systems $(S,\alpha),(T,\beta)$ is given by a function $f \colon S \to T$ such that the following diagram commutes

$$\begin{tikzcd}
S \arrow[r, "f"] \arrow[d, "\alpha^n"'] & T \arrow[d, "\beta^n"] \\
S \arrow[r, "f"] & T
\end{tikzcd}$$

that is, $f \circ \alpha = \beta \circ f$. A natural notion of homomorphism has arisen from the categorical interpretation of the structure.

A further generalization is now possible, if we want to consider continuously-evolving dynamical systems, we can define functors from the monoid of real numbers under adition instead of naturals, that is, considering functors $\mathbb{R} \to \mathsf{Set}$. Note that these functors are given by a set S and a family of morphisms $\{\alpha_r\}_{r \in \mathbb{R}}$ such that

$$\alpha_r \circ \alpha_s = \alpha_{r+s} \qquad \forall r, s \in \mathbb{R}.$$

This example is described in [LS09].

13.4 comma categories

The idea of functor categories leads us to think about categories whose objects are themselves diagrams on a category. The most relevant examples, which will be useful in our development of categorical logic, are the **comma categories**, and specially the particular case of a **slice category**.

Definition 56 (Comma category)**.** Let $\mathcal{C}, \mathcal{D}, \mathcal{E}$ be categories with functors $T \colon \mathcal{E} \to \mathcal{C}$ and $S \colon \mathcal{D} \to \mathcal{C}$. The **comma category** $(T \downarrow S)$ has

- morphisms of the form $f\colon Te \to Sd$ as objects, for $e \in \mathcal{E}, d \in \mathcal{D}$;
- and pairs $\langle k, h\rangle\colon f \to f'$, where $k\colon e \to e'$ and $h\colon d \to d'$ such that $f' \circ Tk = Sh \circ f$, as arrows.

Diagramatically, a morphism in this category is a commutative diagram

$$\begin{tikzcd}
Te \arrow[r, "Tk"] \arrow[d, "f"'] & Te' \arrow[d, "f'"] \\
Sd \arrow[r, "Sh"] & Sd'
\end{tikzcd}$$

where the objects of the category are drawn in grey.

Definition 57 (Slice category)**.** A **slice category** is a particular case of a comma category $(T \downarrow S)$ in which $T = \mathrm{Id}$ is the identity functor and S is a functor from the terminal category, a category with only one object and its identity morphism.

A functor from the terminal category simply chooses an object of the category. If we call $a = S(*)$, objects of this category are morphisms $f\colon c \to a$, where $c \in \mathcal{C}$; and morphisms are $\langle k\rangle\colon f \to f'$, where $k\colon c \to c'$ such that $f' \circ k = f$. Diagramatically a morphism is drawn as

$$\begin{tikzcd}
c \arrow[rr, "Tk"] \arrow[dr, "f"'] & & c' \arrow[dl, "f'"] \\
& a &
\end{tikzcd}$$

This slice category is conventionally written as $(\mathcal{C} \downarrow a)$. In general, we write $(T \downarrow a)$ when S is a functor from the terminal category picking an object a; and we write $(\mathcal{C} \downarrow S)$ when T is the identity functor.

Definition 58 (Coslice category)**.** **Coslice categories** are the categorical dual of slice categories. It is the particular case of a comma category $(T \downarrow S)$ in which $S = \mathrm{Id}$ is the identity functor and T is a functor from the terminal category, a category with only one object and its identity morphism.

If we call $a = T(*)$, objects of this category are morphisms $f\colon c \to a$, where $c \in \mathcal{C}$; and morphisms are $\langle k\rangle\colon f \to f'$, where $k\colon c \to c'$ such that $k \circ f' = f$. Diagramatically a morphism is drawn as

$$\begin{tikzcd}
& a \arrow[dl, "f'"'] \arrow[dr, "f"] & \\
c \arrow[rr, "Sk"] & & c'
\end{tikzcd}$$

This slice category is conventionally written as $(a \downarrow \mathcal{C})$. In general, we write $(a \downarrow S)$ when T is a functor from the terminal category picking an object a; and we write $(T \downarrow \mathcal{C})$ when S is the identity functor.

Definition 59 (Arrow category)**. Arrow categories** are a particular case of comma categories $(T \downarrow S)$ in which both functors are the identity. They are usually written as $\mathcal{C}^{\rightarrow}$.

Objects in this category are morphisms in $\mathcal{C}$, and morphisms in this category are commutative squares in $\mathcal{C}$. Diagramatically,

$$\begin{tikzcd}
a \arrow[r, "k"] \arrow[d, "f"'] & b \arrow[d, "f'"] \\
a' \arrow[r, "h"] & b'
\end{tikzcd}$$

.

14

UNIVERSALITY AND LIMITS

14.1 universal arrows

A **universal property** is commonly given in mathematics by some conditions of existence and uniqueness on morphisms, representing some sort of natural isomorphism. They can be used to define certain constructions up to isomorphism and to operate with them in an abstract setting. We will formally introduce universal properties using *universal arrows* from an object c to a functor S; the property of these arrows is that every arrow of the form $c \to Sd$ will factor uniquely through the universal arrow.

Definition 60 (Universal arrow)**.** A **universal arrow** from c to S is a morphism $u\colon c \to Sr$ such that for every $g\colon c \to Sd$ exists a unique morphism $f\colon r \to d$ making this diagram commute

$$\begin{tikzcd}
 & Sd & d \\
c \arrow[ru, "g"] \arrow[r, "u"'] & Sr \arrow[u, dashed, "Sf"'] & r \arrow[u, dashed, "\exists! f"'] & .
\end{tikzcd}$$

Note how an universal arrow is, equivalently, the initial object of the comma category $(c \downarrow S)$. Thus, universal arrows must be unique up to isomorphism.

Proposition 9 (Universality in terms of hom-sets)**.** *The arrow $u\colon c \to Sr$ is universal if and only if $f \mapsto Sf \circ u$ is a bijection $\hom(r,d) \cong \hom(c, Sd)$ natural in d. Any natural bijection of this kind is determined by a unique universal arrow.*

Proof. On the one hand, given an universal arrow, bijectivity follows from the definition of universal arrow; and naturality follows from the fact that $S(gf) \circ u = Sg \circ Sf \circ u$.

On the other hand, given a bijection φ, we define $u = \varphi(\mathrm{id}_r)$. By naturality, we have the bijection $\varphi(f) = Sf \circ u$, and every arrow is written in this way. □

The categorical dual of an universal arrow from an object to a functor is the notion of universal arrow from a functor to an object. Note how, particularly in this case, we avoid the name *couniversal arrow*; as both arrows are representing what we usually call a *universal property*.

Definition 61 (Dual universal arrow)**.** A **universal arrow** from S to c is a morphism $v\colon Sr \to c$ such that for every $g\colon Sd \to c$ exists a unique morphism $f\colon d \to r$ making this diagram commute

$$\begin{tikzcd}
d \arrow[d, dashed, "\exists! f"'] & Sd \arrow[d, dashed, "Sf"'] \arrow[dr, "g"] & \\
r & Sr \arrow[r, "v"'] & c
\end{tikzcd}$$

14.2 REPRESENTABILITY

Definition 62 (Representation of a functor)**.** A **representation** of a functor from an arbitrary category to the category of sets, $K\colon D \to \mathsf{Set}$, is a natural isomorphism making it isomorphic to a partially applied hom-functor,

$$\psi\colon \hom_D(r,-) \cong K.$$

A functor is *representable* if it has a representation. The object r is called a *representing object*. Note that, for this definition to work, D must have small hom-sets.

Proposition 10 (Representations in terms of universal arrows)**.** *If $u\colon * \to Kr$ is a universal arrow for a functor $K\colon D \to \mathsf{Set}$, then $f \mapsto K(f)(u*)$ is a representation. Every representation is obtained in this way.*

Proof. We know that $\hom(*, X) \dot{\to} X$ is a natural isomorphism in X; in particular $\hom(*, K-) \dot{\to} K-$. Every representation is built then as

$$\hom_D(r,-) \cong \hom(*, K-) \cong K,$$

for every natural isomorphism $D(r,-) \cong \mathsf{Set}(*, K-)$. But every natural isomorphism of this kind is universal arrow. □

14.3 YONEDA LEMMA

Lemma 7 (Yoneda Lemma)**.** *For any $K\colon D \to \mathsf{Set}$ and $r \in D$, there is a bijection*

$$y\colon \mathrm{Nat}(\hom_D(r,-), K) \cong Kr$$

sending the natural transformation $\alpha\colon \hom_D(r,-) \dot{\to} K$ to the image of the identity, $\alpha_r(\mathrm{id}_r)$.

Proof. The complete natural transformation α is determined by $\alpha_r(\mathrm{id}_r)$. By naturality, given any $f\colon r \to s$,

$$\begin{tikzcd}
\hom(r,r) \arrow[rrr, "\alpha_r"] \arrow[ddd, "f\circ -"'] & & & Kr \arrow[ddd, "Kf"] \\
& \mathrm{id}_r \arrow[r, mapsto] \arrow[d, mapsto] & \alpha_r(\mathrm{id}_r) \arrow[d, mapsto] & \\
& f \arrow[r, mapsto] & \alpha_s(f) & \\
\hom(r,s) \arrow[rrr, "\alpha_s"'] & & & Ks
\end{tikzcd}$$

it must be the case that $\alpha_s(f) = Kf(\alpha_r(\mathrm{id}_r))$. □

Corollary 3 (Characterization of natural transformations between representable functors)**.** *Given $r, s \in D$, any natural transformation $\hom(r,-) \overset{.}{\to} \hom(s,-)$ is of the form $-\circ h$ for a unique morphism $h\colon s \to r$.*

Proof. Using Yoneda Lemma (Lemma 7), we know that

$$\mathrm{Nat}(\hom_D(r,-), \hom_D(s,-)) \cong \hom_D(s,r),$$

sending the natural transformation to a morphism $\alpha(id_r) = h\colon s \to r$. The rest of the natural transformation is determined as $-\circ h$ by naturality. □

Proposition 11 (Naturality of the Yoneda Lemma)**.** *The bijection on the Yoneda Lemma (Lemma 7),*

$$y\colon \mathrm{Nat}(\hom_D(r,-), K) \cong Kr,$$

is a natural isomorphism between two functors from $\mathsf{Set}^D \times D$ *to* Set.

Proof. We define $N\colon \mathsf{Set}^D \times D \to \mathsf{Set}$ on objects as $N\langle r, K\rangle = \mathrm{Nat}(\hom(r,-), K)$. Given $f\colon r \to r'$ and $F\colon K \overset{.}{\to} K'$, the functor is defined on morphisms as

$$N\langle f, F\rangle(\alpha) = F \circ \alpha \circ (-\circ f) \in \mathrm{Nat}(\hom(r',-), K),$$

where $\alpha \in \mathrm{Nat}(\hom(r,-), K)$. We define $E\colon \mathsf{Set}^D \times D \to \mathsf{Set}$ on objects as $E\langle r, K\rangle = Kr$. Given $f\colon r \to r'$ and $F\colon K \overset{.}{\to} K'$, the functor is defined on morphisms as

$$E\langle f, F\rangle(a) = F(Kf(a)) = K'f(Fa) \in K'r',$$

where $a \in Kr$, and the equality holds because of the naturality of F. The naturality of y is equivalent to the commutativity of the following diagram

$$\begin{tikzcd}
\mathrm{Nat}(\hom(r,-), K) \arrow[r, "y"] \arrow[d, "N\langle f,F\rangle"'] & Kr \arrow[d, "E\langle f,F\rangle"] \\
\mathrm{Nat}(\hom(r',-), K') \arrow[r, "y"] & K'r'
\end{tikzcd}$$

where, given any $\alpha \in \mathrm{Nat}(\hom(r,-),K)$, it follows from naturality of α that

$$\begin{aligned} y\big(N\langle f,F\rangle(\alpha)\big) &= y\big(F\circ\alpha\circ(-\circ f)\big) = F\circ\alpha\circ(-\circ f)(\mathrm{id}_{r'}) = F(\alpha(f)) \\ &= F(\alpha(\mathrm{id}_{r'}\circ f)) = F(Kf(\alpha_r(\mathrm{id}_r))) = E\langle f,F\rangle\alpha_r(\mathrm{id}_r) \\ &= E\langle f,F\rangle\big(y(\alpha)\big). \end{aligned}$$

□

Definition 63. In the conditions of the Yoneda Lemma (Lemma 7) the **Yoneda functor**, $Y\colon D^{op}\to \mathsf{Set}^D$, is defined with the arrow function

$$(f\colon s\to r)\mapsto \Big(\hom_D(f,-)\colon\ \hom_D(r,-)\to\hom_D(s,-)\Big).$$

It can be also written as $Y'\colon D\to\mathsf{Set}^{D^{op}}$.

Proposition 12. *The Yoneda functor is full and faithful.*

Proof. By Yoneda Lemma, we know that

$$y\colon\ \mathrm{Nat}(\hom(r,-),\hom(s,-))\cong\hom(s,r)$$

is a bijection, where $y(\hom(f,-))=f$. □

14.4 LIMITS

In the definition of product, we chose two objects of the category, we considered all possible ***cones*** over two objects and we picked the universal one. Diagramtically,

$$\begin{tikzcd} & c \arrow[dl] \arrow[dr] & \\ a & & b \end{tikzcd} \qquad \begin{tikzcd} & d \arrow[d, dashed, "\exists!"] \arrow[ddl, bend right] \arrow[ddr, bend left] & \\ & a\times b \arrow[dl] \arrow[dr] & \\ a & & b \end{tikzcd}$$

c is a cone and $a\times b$ is the universal one: every cone factorizes through it. In this particular case, the base of each cone is given by two objects; or, in other words, by the image of a functor from the discrete category with only two objects, called the *index category*.

We will be able to create new constructions on categories by formalizing the notion of cone and generalizing to arbitrary bases, given as functors from arbitrariliy complex index categories. Constant functors are the first step into formalizing the notion of cone.

Definition 64 (Constant functor)**.** The **constant functor** $\Delta\colon \mathcal{C} \to \mathcal{C}^{\mathcal{J}}$ sends each object $c \in \mathcal{C}$ to a constant functor $\Delta c\colon \mathcal{J} \to \mathcal{C}$ defined as

- the constantly-c function for objects, $\Delta(j) = c$;
- and the constantly-id_c function for morphisms, $\Delta(f) = \mathrm{id}_c$.

The constant functor sends a morphism $g\colon c \to c'$ to a natural transformation $\Delta g\colon \Delta c \to \Delta c'$ whose components are all g.

We could say that Δc squeezes the whole category $\mathcal{J}$ into c. A natural transformation from this functor to some other $F\colon \mathcal{J} \to \mathcal{C}$ should be regarded as a ***cone*** from the object c to a copy of $\mathcal{J}$ inside the category $\mathcal{C}$; as the following diagram exemplifies

$$\begin{tikzcd}
 & & & & & & c \arrow[d] \arrow[dr, bend left] \arrow[ddl, bend right] \arrow[ddrr, bend left] & & \\
 & \cdot \arrow[r] \arrow[dl] \arrow[drr] & \cdot \arrow[dr] & & \arrow[r, Rightarrow, "\Delta"] & {} & \cdot \arrow[r] \arrow[dl] \arrow[drr] & \cdot \arrow[dr] & \\
\cdot \arrow[rrr] & & & \cdot & \arrow[r, Rightarrow, "F"'] & \cdot \arrow[rrr] & & & \cdot \\
 & & \mathcal{J} & & & & & F\mathcal{J} &
\end{tikzcd}$$

The components of the natural transformation appear highlighted in the diagram. The naturality of the transformation implies that each triangle

$$\begin{tikzcd}
 & \cdot \arrow[dl] \arrow[dr] & \\
\cdot \arrow[rr] & & \cdot
\end{tikzcd}$$

on that cone must be commutative. Thus, natural transformations are a way to recover all the information of an arbitrary ***index category*** $\mathcal{J}$ that was encoded in c by the constant functor. As we did with products, we want to find the cone that best encodes that information; a universal cone, such that every other cone factorizes through it. Diagramatically an r such that, for each d,

$$\begin{tikzcd}
 & d \arrow[d, dashed, "\exists!"] \arrow[dd, bend right] \arrow[ddr, bend left] \arrow[dddl, bend right=50] \arrow[dddrr, bend left=50] & & \\
 & r \arrow[d] \arrow[dr, bend left] \arrow[ddl, bend right] \arrow[ddrr, bend left] & & \\
 & \cdot \arrow[r] \arrow[dl] \arrow[drr] & \cdot \arrow[dr] & \\
\cdot \arrow[rrr] & & & \cdot
\end{tikzcd}$$

That factorization will be represented in the formal definition of limit by a universal natural transformation between the two constant functors.

Definition 65 (Limit)**.** The **limit** of a functor $F\colon \mathcal{J} \to \mathcal{C}$ is an object $r \in \mathcal{C}$ such that there exists a universal arrow $v\colon \Delta r \xrightarrow{\cdot} F$ from Δ to F. It is usually written as $r = \varprojlim F$.

That is, for every natural transformation $w\colon \Delta d \xrightarrow{\cdot} F$, there is a unique morphism $f\colon d \to r$ such that

$$\begin{tikzcd} d \arrow[d, dashed, "\exists! f"'] & \Delta d \arrow[d, dashed, "\Delta f"'] \arrow[dr, "w"] & \\ r & \Delta r \arrow[r, "v"'] & F \end{tikzcd}$$

commutes. This reflects directly on the universality of the cone we described earlier and proves that limits are unique up to isomorphism.

By choosing different index categories, we will be able to define multiple different constructions on categories as limits.

14.5 EXAMPLES OF LIMITS

For our first example, we will take the following category, called $\downdownarrows$ as index category,

$$\begin{tikzcd} \cdot \arrow[r, bend left] \arrow[r, bend right] & \cdot \end{tikzcd}$$

A functor $F\colon\ \downdownarrows \to \mathcal{C}$ is a pair of parallel arrows in $\mathcal{C}$. Limits of functors from this category are called **equalizers**. With this definition, the **equalizer** of two parallel arrows $f, g\colon a \to b$ is an object $\mathrm{eq}(f,g)$ with a morphism $e\colon \mathrm{eq}(f,g) \to a$ such that $f \circ e = g \circ e$; and such that any other object with a similar morphism factorizes uniquely through it

$$\begin{tikzcd} & d \arrow[d, dashed, "\exists!"] \arrow[ddl, bend right, "e'"'] \arrow[ddr, bend left] & \\ & \mathrm{eq}(f,g) \arrow[dl, "e"'] \arrow[dr] & \\ a \arrow[rr, bend left, "f"] \arrow[rr, bend right, "g"'] & & b \end{tikzcd}$$

note how the right part of the cone is completely determined as $f \circ e$. Because of this, equalizers can be written without specifying it, and the diagram can be simplified to

$$\begin{tikzcd} \mathrm{eq}(f,g) \arrow[r, "e"] & a \arrow[r, bend left, "f"] \arrow[r, bend right, "g"'] & b \\ d \arrow[u, dashed, "\exists!"] \arrow[ur, "e'"'] & & \end{tikzcd}$$

.

Example 19 (Equalizers in Sets). The equalizer of two parallel functions $f, g\colon A \to B$ in Set is $\{x \in A \mid f(a) = g(a)\}$ with the inclusion morphism. Given any other function

$h\colon D \to A$ such that $f \circ h = g \circ h$, we know that $f(h(d)) = g(h(d))$ for any $d \in D$. Thus, h can be factorized through the equalizer.

$$\begin{tikzcd}
\{x \in A \mid f(a) = g(a)\} \arrow[r, hook, "i"] & A \arrow[r, bend left, "f"] \arrow[r, bend right, "g"'] & B \\
D \arrow[u, dashed, "\exists!"] \arrow[ru, "h"'] & &
\end{tikzcd}$$

Example 20 (Kernels). In the category of abelian groups, the kernel of a function f, $\ker(f)$, is the equalizer of $f\colon G \to H$ and a function sending each element to the zero element of H. The same notion of kernel can be defined in the category of R-Modules, for any ring R.

Pullbacks are defined as limits whose index category is $\cdot \to \cdot \leftarrow \cdot$. Any functor from that category is a pair of arrows with a common codomain; and the pullback is the universal cone over them.

$$\begin{tikzcd}
& d \arrow[d, dashed, "\exists!"] \arrow[ddl, bend right, "p'"'] \arrow[ddr, bend left, "q'"] \arrow[dd, bend right] & \\
& a \arrow[dl, bend right, "p"'] \arrow[dr, bend left, "q"] \arrow[d] & \\
x \arrow[r, "f"'] & z & y \arrow[l, "g"]
\end{tikzcd}$$

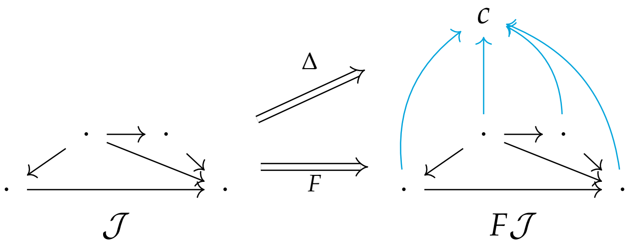
Again, the central arrow of the diagram is determined as $f \circ q = g \circ p$; so it can be ommited in the diagram. The usual definition of a pullback for two morphisms $f\colon x \to z$ and $g\colon y \to z$ is pair of morphisms $p\colon a \to x$ and $q\colon a \to y$ such that $f \circ q = g \circ p$ which are also universal, that is, given any pair of morphisms $p'\colon d \to x$ and $q'\colon d \to y$, there exists a unique $u\colon d \to a$ making the diagram commute. Usually we write the pullback object as $x \times_z y$ and we write this property diagramatically as

$$\begin{tikzcd}
d \arrow[rrd, bend left, "p'"] \arrow[rdd, bend right, "q'"'] \arrow[rd, dashed, "\exists! u"] & & \\
& x \times_z y \arrow[r, "p"] \arrow[d, "q"'] & x \arrow[d, "f"] \\
& y \arrow[r, "g"'] & z
\end{tikzcd}$$

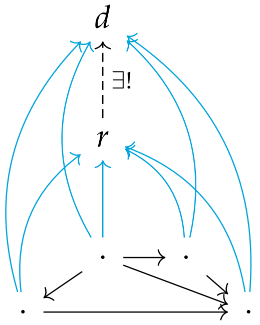
The square in this diagram is usually called a *pullback square*, and the pullback object is usually called a *fibered product*.

14.6 colimits

A colimit is the dual notion of a limit. We could consider cocones to be the dual of cones and pick the universal one. Once an index category $\mathcal{J}$ and a base category $\mathcal{C}$ are fixed, a *cocone* is a natural transformation from a functor on the base category to a constant functor. Diagramatically,



is an example of a cocone, and the universal one would be the r, such that, for each cone d,



and naturality implies that each triangle

$$\begin{tikzcd}
 & \cdot & \\
\cdot \arrow[ur] \arrow[rr] & & \cdot \arrow[ul]
\end{tikzcd}$$

commutes.

Definition 66 (Colimits)**.** The **colimit** of a functor $F\colon J \to \mathcal{C}$ is an object $r \in \mathcal{C}$ such that there exists a universal arrow $u\colon F \xrightarrow{\cdot} \Delta r$ from F to Δ. It is usually written as $r = \varinjlim F$.

That is, for every natural transformation $w\colon F \xrightarrow{\cdot} \Delta d$, there is a unique morphism $f\colon r \to d$ such that

$$\begin{tikzcd}
d & \Delta d & \\
r \arrow[u, dashed, "\exists! f"] & \Delta r \arrow[u, dashed, "\Delta f"] & F \arrow[l, "v"] \arrow[ul, "w"']
\end{tikzcd}$$

commutes. This reflects directly on the universality of the cocone we described earlier and proves that colimits are unique up to isomorphism.

14.7 EXAMPLES OF COLIMITS

Coequalizers are the dual of *equalizers*; colimits of functors from $\downdownarrows$. The coequalizer of two parallel arrows is an object $\mathrm{coeq}(f,g)$ with a morphism $e\colon b \to \mathrm{coeq}(f,g)$ such that $e \circ f = e \circ g$; and such that any other object with a similar morphism factorizes uniquely through it

$$\begin{tikzcd}
 & d & \\
 & \mathrm{coeq}(f,g) \arrow[u, dashed, "\exists!"] & \\
a \arrow[ur] \arrow[uur, bend left] \arrow[rr, bend left, "f"] \arrow[rr, bend right, "g"'] & & b \arrow[ul, "e"'] \arrow[uul, bend right, "e'"']
\end{tikzcd}$$

as the right part of the cocone is completely determined by the left one, the diagram can be written as

$$\begin{tikzcd}
a \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & b \arrow[r, "e"] \arrow[dr, "e'"'] & \mathrm{coeq}(f,g) \arrow[d, dashed, "\exists!"] \\
 & & d
\end{tikzcd}$$

Example 21 (Coequalizers in Sets). The coequalizer of two parallel functions $f,g\colon A \to B$ in Set is $B/(\sim_{f=g})$, where $\sim_{f=g}$ is the minimal equivalence relation in which we have $f(a) \sim g(a)$ for each $a \in A$. Given any other function $h\colon B \to D$ such that $h(f(a)) = h(g(a))$, it can be factorized in a unique way by $h'\colon B/\sim_{f=g} \to D$.

$$\begin{tikzcd}
A \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & B \arrow[r, "e"] \arrow[dr, "e'"'] & B/(\sim_{f=g}) \arrow[d, dashed, "\exists!"] \\
 & & D
\end{tikzcd}$$

Pushouts are the dual of pullbacks; colimits whose index category is $\cdot \leftarrow \cdot \rightarrow \cdot$, that is, the dual of the index category for pullbacks. Diagramatically,

$$\begin{tikzcd}
 & d & \\
 & a \arrow[u, dashed, "\exists!"] & \\
x \arrow[ur, bend left, "p"] \arrow[uur, bend left, "p'"] & z \arrow[l, "f"] \arrow[r, "g"'] \arrow[u] \arrow[uu, bend left] & y \arrow[ul, bend right, "q"'] \arrow[uul, bend right, "q'"']
\end{tikzcd}$$

and we can define the pushout of two morphisms $f\colon z \to x$ and $g\colon z \to y$ as a pair of morphisms $p\colon x \to a$ and $q\colon y \to a$ such that $p \circ f = q \circ g$ which are also universal, that is, given any pair of morphisms $p' \colon x \to d$ and $q' \colon y \to d$, there exists a unique $u \colon a \to d$ making the diagram commute.

$$\begin{tikzcd}
z \arrow[r, "g"] \arrow[d, "f"'] & y \arrow[d, "q"] \arrow[ddr, bend left, "q'"] & \\
x \arrow[r, "p"] \arrow[drr, bend right, "p'"'] & x \amalg_z y \arrow[dr, dashed, "\exists ! u"'] & \\
 & & d
\end{tikzcd}$$

The square in this diagram is usually called a *pushout square*, and the pullback object is usually called a *fibered coproduct*.

15

ADJOINTS, MONADS AND ALGEBRAS

15.1 ADJUNCTIONS

Definition 67 (Adjunction)**.** An **adjunction** from categories $\mathcal{X}$ to $\mathcal{Y}$ is a pair of functors $F\colon \mathcal{X} \to \mathcal{Y}$, $G\colon \mathcal{Y} \to \mathcal{X}$ with a natural bijection

$$\varphi\colon\ \hom(Fx, y) \cong \hom(x, Gy),$$

natural in both $x \in \mathcal{X}$ and $y \in \mathcal{Y}$. We write it as $F \dashv G$.

Naturality of φ means that both

$$\begin{tikzcd}
\hom(Fx,y) \arrow[r, "\varphi_{x,y}"] \arrow[d, swap, "-\circ Fh"] & \hom(x,Gy) \arrow[d, "-\circ h"] \\
\hom(Fx',y) \arrow[r, swap, "\varphi_{x,y}"] & \hom(x',Gy)
\end{tikzcd}
\qquad
\begin{tikzcd}
\hom(Fx,y) \arrow[r, "\varphi_{x,y}"] \arrow[d, swap, "k\circ -"] & \hom(x,Gy) \arrow[d, "Gk\circ -"] \\
\hom(Fx,y') \arrow[r, swap, "\varphi_{x,y}"] & \hom(x,Gy')
\end{tikzcd}$$

commute for every $h\colon x \to x'$ and $k\colon y \to y'$. That is, for every $f\colon Fx \to y$, $\varphi(f) \circ h = \varphi(f \circ Fh)$ and $Gk \circ \varphi(f) = \varphi(k \circ f)$. Equivalently, φ^{-1} is natural and that means that, for every $g\colon x \to Gy$, $k \circ \varphi^{-1}(g) = \varphi^{-1}(Gk \circ g)$ and $\varphi^{-1}(g) \circ Fh = \varphi^{-1}(g \circ h)$.

In the following two propositions, we will characterize all this information in terms of natural transformations made up of universal arrows.

Proposition 13 (Unit and counit of an adjunction)**.** *An adjunction determines a **unit** and a **counit**;*

1. *the **unit** is natural transformation made with universal arrows $\eta\colon \mathrm{Id} \xrightarrow{\cdot} GF$, where the right adjunct of each $f\colon Fx \to y$ is*

$$\varphi f = Gf \circ \eta_x\colon x \to Gy;$$

2. *the **counit** is natural transformation made with universal arrows $\varepsilon\colon FG \xrightarrow{\cdot} \mathrm{Id}$, where the left adjunct of each $g\colon x \to Gy$ is*

$$\varphi^{-1} g = \varepsilon \circ Fg\colon Fx \to y;$$

that follow the ***triangle identities*** $G\varepsilon \circ \eta G = \mathrm{id}_G$ *and* $\varepsilon F \circ F\eta = \mathrm{id}_F$.

Proof. We define $\eta_x = \varphi(\mathrm{id}_{Fx})$ and $\varepsilon_x = \varphi^{-1}(\mathrm{id}_{Gx})$. The right and left adjunct formulas and the naturality of η and ε can be deduced from the naturality of φ,

$$\begin{aligned} GFh \circ \eta_x = GFh \circ \varphi(\mathrm{id}_{Fx}) = \varphi(Fh) = \varphi(\mathrm{id}_{Fx}) \circ h = \eta_x \circ h; \\ \varepsilon_x \circ FGk = \varphi^{-1}(\mathrm{id}_{Fx}) \circ FGk = \varphi^{-1}(Gk) = k \circ \varphi^{-1}(\mathrm{id}_{Gx}) = k \circ \varepsilon_x. \end{aligned}$$

Finally, the triangle identities also follow from naturality of φ,

$$\begin{aligned} \mathrm{id} &= \varphi(\varepsilon) = G\varepsilon \circ \eta; \\ \mathrm{id} &= \varphi^{-1}(\eta) = \varepsilon \circ F\eta. \end{aligned}$$

□

Proposition 14 (Characterization of adjunctions). *Each adjunction is completely determined by any of the following data,*

1. *functors* F, G *and* $\eta\colon 1 \dot\to GF$ *where* $\eta_x\colon x \to GFx$ *is universal to* G.
2. *functor* G *and universals* $\eta_x\colon x \to GF_0x$, *where* $F_0x \in \mathcal{Y}$.
3. *functors* F, G *and* $\varepsilon\colon FG \dot\to 1$ *where* $\varepsilon_a\colon FGa \to a$ *is universal from* F.
4. *functor* F *and universals* $\varepsilon_a\colon FG_0a \to a$, *creating a functor* G.
5. *functors* F, G, *with natural transformations satisfiying the triangle identities* $G\varepsilon \circ \eta G = \mathrm{id}$ *and* $\varepsilon F \circ F\eta = \mathrm{id}$.

Proof.
1. Universality of η_x gives a isomorphism $\varphi\colon \hom(Fx, y) \cong \hom(x, Gy)$ between the arrows in the following diagram

$$\begin{tikzcd} & Gy & y \\ x \arrow[ur, "f"] \arrow[r, "\eta_x"'] & GFx \arrow[u, dashed, "Gg"'] & Fx \arrow[u, dashed, "\exists!g"'] \end{tikzcd}$$

   defined as $\varphi(g) = Gg \circ \eta_x$. This isomorphism is natural in x; for every $h\colon x' \to x$ we know by naturality of η that $Gg \circ \eta \circ h = G(g \circ Fh) \circ \eta$. The isomorphism is also natural in y; for every $k\colon y \to y'$ we know by functoriality of G that $Gh \circ Gg \circ \eta = G(h \circ g) \circ \eta$.
2. We can define a functor F on objects as $Fx = F_0x$. Given any $h\colon x \to x'$, we can use the naturality of η to define Fh as the unique arrow making this diagram commute

$$\begin{tikzcd} & GFx' & Fx' \\ x \arrow[ur, "\eta_{x'} \circ h"] \arrow[r, "\eta_x"'] & GFx \arrow[u, dashed, "GFh"'] & Fx \arrow[u, dashed, "\exists!Fh"'] \end{tikzcd}$$

and this choice makes F a functor, as it can be checked in the following diagrams using the existence and uniqueness given by the universality of η in both cases

$$\begin{tikzcd}
 & & & x'' \arrow[r, "\eta_{x''}"] & GFx'' & Fx'' \\
 & GFx & Fx & x' \arrow[u, "h'"] \arrow[r, "\eta_{x'}"] & GFx' \arrow[u, "GFh'"'] & Fx' \arrow[u, dashed, "\exists!Fh'"] \\
x \arrow[ur, "\eta_x"] \arrow[r, "\eta_x"'] & GFx \arrow[u, dashed, "\mathrm{id}"'] & Fx \arrow[u, dashed, "\mathrm{id}"'] & x \arrow[u, "h"] \arrow[r, "\eta_x"] & GFx \arrow[u, "GFh"'] & Fx \arrow[u, dashed, "\exists!Fh'"] \arrow[uu, dashed, bend right, "\exists!F(h'\circ h)"']
\end{tikzcd}$$

3. The proof is dual to that of 1.
4. The proof is dual to that of 2.
5. We can define two functions $\varphi(f) = Gf \circ \eta_x$ and $\theta(g) = \varepsilon_y \circ Fg$. We checked in 1 (and 3) that these functions are natural in both arguments; now we will see that they are inverses of each other using naturality and the triangle identities

$$\begin{aligned}
\varphi(\theta(g)) &= G\varepsilon_a \circ GFg \circ \eta_x = G\varepsilon_a \circ \eta_x \circ g = g;\\
\theta(\varphi(f)) &= \varepsilon \circ FGf \circ F\eta = f \circ \varepsilon \circ F\eta = f.
\end{aligned}$$

□

15.2 properties of adjoints

Proposition 15 (Essential uniqueness of adjoints). *Two adjoints to the same functor $F, F' \dashv G$ are naturally isomorphic.*

Proof. □

Theorem 9 (Composition of adjunctions). *Given two adjunctions between categories $\mathcal{X}, \mathcal{Y}$ and $\mathcal{Y}, \mathcal{Z}$ respectively,*

$$\varphi\colon\ \hom(Fx, y) \cong \hom(x, Gy) \qquad \theta\colon\ \hom(F'y, z) \cong \hom(G'y, z)$$

the composite functors yield a composite adjunction

$$\theta \cdot \varphi\colon\ \hom(F'Fx, y) \cong \hom(x, GG'z).$$

If the unit and counit of φ are $\langle \eta, \varepsilon \rangle$ and the unit and counit of θ are $\langle \eta', \varepsilon' \rangle$; the unit and counit of the composite adjunction are $\langle G\eta' F \circ \eta,\ \varepsilon' \circ F'\varepsilon G' \rangle$.

Proof. □

15.3 MONADS

Definition 68 (Monad). A **monad** is a functor $T\colon X \to X$ with natural transformations

- $\eta\colon I \overset{\cdot}{\to} T$, called *unit*
- $\mu\colon T^2 \overset{\cdot}{\to} T$, called *multiplication*

such that

$$\begin{tikzcd}
T^3 \arrow[r, "T\mu"] \arrow[d, "\mu T"] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"] & T
\end{tikzcd}
\qquad
\begin{tikzcd}
IT \arrow[r, "\eta T"] \arrow[dr, "\cong"'] & T^2 \arrow[d, "\mu"] & TI \arrow[l, "T\eta"'] \arrow[dl, "\cong"] \\
 & T &
\end{tikzcd}
.$$

Proposition 16 (Each adjunction gives rise to a monad). *Given $F \dashv G$, GF is a monad.*

Proof. We take the unit of the adjunction as the monad unit. We define the product as $\mu = G\varepsilon F$. Associativity follows from these diagrams

$$\begin{tikzcd}
FGFG \arrow[r, "FG\varepsilon"] \arrow[d, "\varepsilon FG"'] & FG \arrow[d, "\varepsilon"] \\
FG \arrow[r, "\varepsilon"] & I
\end{tikzcd}
\qquad
\begin{tikzcd}
GFGFGF \arrow[r, "GFG\varepsilon F"] \arrow[d, "G\varepsilon FGF"'] & GFGF \arrow[d, "G\varepsilon F"] \\
GFGF \arrow[r, "G\varepsilon F"] & GF
\end{tikzcd}
,$$

where the first is commutative by Proposition [BROKEN LINK: prop-interchangelaw] and the second is obtained by applying functors G and F. Unit laws follow from the [BROKEN LINK: *Unit and counit] after applying F and G. □

Definition 69 (Comonad). A **comonad** is a functor $L\colon X \to X$ with natural transformations

- $\varepsilon\colon L \to I$, called *counit*
- $\delta\colon L \to L^2$, called *comultiplication*

such that

$$\begin{tikzcd}
L \arrow[r, "\delta"] \arrow[d, "\delta"'] & L^2 \arrow[d, "L\delta"] \\
L^2 \arrow[r, "\delta L"] & L^3
\end{tikzcd}
\qquad
\begin{tikzcd}
 & L \arrow[dl, "\cong"'] \arrow[d, "\delta"] \arrow[dr, "\cong"] & \\
IL & L^2 \arrow[l, "\varepsilon L"] \arrow[r, "L\varepsilon"'] & LI
\end{tikzcd}
.$$

15.4 algebras for a monad

Definition 70 (T-algebra)**.** For a monad T, a T**-algebra** is an object x with an arrow $h\colon Tx \to x$ called *structure map* making these diagrams commute

$$\begin{tikzcd}
T^2x \arrow[r, "Th"] \arrow[d, "\mu"'] & Tx \arrow[d, "h"] \\
Tx \arrow[r, "h"] & x
\end{tikzcd}$$

Definition 71 (Morphism of T-algebras)**.** A **morphism of T-algebras** is an arrow $f\colon x \to x'$ making the following square commute

$$\begin{tikzcd}
Tx \arrow[r, "h"] \arrow[d, "Tf"'] & Tx \arrow[d, "f"] \\
Tx' \arrow[r, "h'"'] & Tx'
\end{tikzcd}$$

Proposition 17 (Category of T-algebras)**.** *The set of all T-algebras and their morphisms form a category X^T.*

Proof. Given $f\colon x \to x'$ and $g\colon x' \to x''$, T-algebra morphisms, their composition is also a T-algebra morphism, due to the fact that this diagram

$$\begin{tikzcd}
Tx \arrow[r, "h"] \arrow[d, "Tf"'] & x \arrow[d, "f"] \\
Tx' \arrow[r, "h'"] \arrow[d, "Tg"'] & x' \arrow[d, "g"] \\
Tx'' \arrow[r, "h''"] & x''
\end{tikzcd}$$

commutes. □

15.5 free algebras for a monad

15.6 kleisli categories

Definition 72 (Kleisli category)**.** The **Kleisli category** of a monad $T\colon \mathcal{X} \to \mathcal{X}$ is written as $\mathcal{X}_T$ and is given by

- an object x_T for every $x \in \mathcal{X}$; and
- an arrow $f^\flat\colon x_T \to y_T$ for every $f\colon x \to Ty$.

And the composite of two morphisms is defined as

$$g^\flat \circ f^\flat = (\mu \circ Tg \circ f)^\flat.$$

Theorem 10 (Adjunction on a Kleisli category). *The functors $F_T \colon \mathcal{X} \to \mathcal{X}_T$ and $G \colon \mathcal{X}_T \to \mathcal{X}$ from and to the Kleisli category which are defined on objects as $F_T(x) = x_T$ and $G_T(x_T) = Tx$, and defined on morphisms as*

$$\begin{aligned} F_T \colon (k \colon x \to y) &\quad \mapsto (\eta_y \circ k)^\flat \colon x_T \to y_T \\ G_T \colon (f^\flat \colon x_T \to y_T) &\quad \mapsto (\eta_y \circ Tf) \colon Tx \to Ty \end{aligned}$$

form an adjunction $\flat \colon \operatorname{hom}(x, Ty) \to \operatorname{hom}(x_T, y_T)$ whose monad is precisely T.

Proof. □

Part IV

CATEGORICAL LOGIC

This section is based on [MM94].

Topos theory arises independently with Grothendieck and sheaf theory, Lawvere and the axiomatization of set theory and Paul Cohen with the forcing techniques with allowed to construct new models of ZFC.

16

LAWVERE THEORIES

This section follows [AB17].

16.1 MOTIVATION FOR ALGEBRAIC THEORIES

We will develop an unified approach to the study of algebraic structures based on constants, operations and equations; such as groups, modules or rings. Our **algebraic theories** are usually given by

- a *signature*, a family of sets $\{\Sigma_k\}_{k\in\mathbb{N}}$ whose elements are called *k-ary operations*. The *terms* of a signature are defined inductively, being variables or k-ary operations applied to k-tuples of terms;
- and a set of *axioms*, which are equations between terms.

For example, the theory of groups is given by a signature conaining

- a *binary operation* called $\cdot$,
- a *unary operation* written as $^{-1}$, and
- a *nullary operation* or a *constant* called e.

Satisfiying the following axioms

- $(x \cdot y) \cdot z = x \cdot (y \cdot z)$,
- $x \cdot e = x$,
- $e \cdot x = x$,
- $x \cdot x^{-1} = e$, and
- $x^{-1} \cdot x = e$.

Note how quantifiers are not needed here, as we are interpreting each x, y, z as free variables and the universal quantification is therefore implicit. Theories in which the operations are not defined for every possible term, cannot be expressed in this way. Fields, in which the inverse of 0 is not defined, are not expressable in this form; this fact will be proved formally in Example 22.

A theory can be **interpreted** on a suitable category $\mathcal{C}$ as

- an object of the category, $A \in \mathcal{C}$;
- with morphisms $If\colon A^k \to A$ for every k-ary operation f.

Any interpretation of the theory induces an interpretation for every term on a context. That is, a term t can be given in the variable context $x_1, \dots, x_n$ if all variables that appear in t appear in $x_1, \dots, x_n$. We write that as

$$x_1, x_2, \dots, x_n \mid t;$$

and the **interpretation of the term** $x_1, \dots, x_n \mid t$ **on that context** is a morphism $I(x_1, \dots, x_n \mid t)\colon A^n \to A$ defined inductively knowing that

- the interpretation of the i-th variable is the i-th projection

  $$I(x_1, \dots, x_n \mid x_i) = \pi_i : A^n \to A,$$

- the interpretation of an operation over a term is the interpretation of the morphism componsed with the componentwise interpretation of subterms

  $$I(f\langle t_1, \dots, t_k\rangle) = If \circ \langle It_1, \dots, It_k\rangle\colon A^n \to A.$$

  where we implicitely assume the context to be $x_1, \dots, x_n$.

The interpretation of a particular variable depends therefore on the context. We say that an interpretation **satisfies** an equation $\Gamma \mid u = v$ in a particular given context if the interpretation of both terms of the equation is the same on that context, $I(\Gamma \mid u) = I(\Gamma \mid v)$.

We usually would like to find interpretations where all the axioms of the theory were satisfied. These are called **models of the algebraic theory**. The problem with this notion of algebraic theory is that it is not representation-free; it is not independent of the choice of constants, operations or axioms. There may be multiple formulations of the same theory, with different but equivalent axioms. For instance, [Mcc91] discusses many single-equation axiomatizations of groups, such as

$$x \;/\; \big(((x/x)/y)/z\big) \;/\; \big((x/x)/x)/z\big) = y$$

with the binary operation $/$, related to the usual multiplication as $x/y = x \cdot y^{-1}$.

Our solution to this problem will be to capture all the algebraic information of a theory – all operations, constants and axioms – into a category. Differently presented but equivalent theories will give rise to the same category. This category will have *contexts* $[x_1, \dots, x_n]$ as objects. A morphism from $[x_1, \dots, x_n]$ to $[x_1, \dots, x_m]$ will be a tuple of terms

$$\langle t_1, \dots, t_k\rangle\colon [x_1, \dots, x_n] \to [x_1, \dots, x_m]$$

such that every t_k is given in the context $[x_1, \dots, x_n]$. Composition is defined componentwise as substitution of the terms of the first morphism into the variables of the second one, that is,

$$\langle s_1, \dots, s_n\rangle = \langle u_1, \dots, u_n\rangle \circ \langle t_1, \dots, t_m\rangle,$$

where

$$s_i = u_i[t_1, \dots, t_m / x_1, \dots, x_m].$$

Two morphisms in this category $\langle t_1, \dots, t_n \rangle$ and $\langle s_1, \dots, s_n \rangle$ are equal if the axioms of the theory imply the componentwise equality of its terms, that is, $t_i = s_i$.

This interpretation will lead us to our definition of **algebraic theory** as a category with finite products.

Every model M in the previous sense could be seen as a functor from this category to a given category $\mathcal{C}$ preserving finite products. Once the image of $M[x_1] = A$ is chosen, the functor is determined on objects by

$$M[x_1, \dots, x_n] = A^k$$

and once it is defined for the basic operations, it is inductively determined on morphisms as

- $M\langle x_i \rangle = \pi_i \colon A^k \to A$, for any morphism $\langle x_i \rangle$;
- $M\langle t_1, \dots, t_m \rangle = \langle Mt_1, \dots, Mt_m \rangle \colon A^m \to A$, the componentwise interpretation of subterms;
- $M\langle f \langle t_1, \dots, t_m \rangle \rangle = Mf \circ \langle Mt_1, \dots, Mt_m \rangle \colon (M\mathbb{A})^m \to M\mathbb{A}$.

The fact that M is a well-defined functor follows from the assumption that it is a model.

16.2 algebraic theories as categories

Definition 73 (Lawvere algebraic theory)**.** An **algebraic theory** is a category $\mathbb{A}$ with finite products and objects forming a sequence $A^0, A^1, A^2, \dots$ such that $A^m \times A^n = A^{m+n}$ for any m, n.

From this definition, it follows that A^0 must be the terminal object.

Definition 74 (Model)**.** A **model** of an algebraic theory $\mathbb{A}$ in a category $\mathcal{C}$ is a functor $M \colon \mathbb{A} \to \mathcal{C}$ preserving all finite products.

Definition 75 (Category of models of a theory)**.** The **category of models** $\mathtt{Mod}_{\mathcal{C}}(\mathbb{A})$ is the full subcategory of functor category $\mathcal{C}^{\mathbb{A}}$ given by the functors preserving all finite products. Morphisms between models of a theory in a category are natural transformations.

Definition 76 (Algebraic category)**.** An **algebraic category** is category equivalent to a category of the form $\mathtt{Mod}_{\mathcal{C}}(\mathbb{A})$, where $\mathbb{A}$ is an algebraic theory.

Example 22 (Fields have no algebraic theory)*.* The category `Fields` is not an algebraic category. Any algebraic category $\mathtt{Mod}_{\mathcal{C}}(\mathbb{A})$ has a terminal object given by the constant functor $\Delta_1 : \mathbb{A} \to \mathcal{C}$ to 1, the terminal object of $\mathcal{C}$. Note that $\mathcal{C}$ must have a terminal object for a model to it to exist, as models must preserve all finite products. We know that Δ_1 is a terminal object because, in general, it is the terminal object of the category of functors $\mathcal{C}^{\mathbb{A}}$. However, `Fields` has no terminal object.

16.3 completeness for algebraic theories

When defining interpretation of algebraic theories as categories we should ensure the property of ***semantic completeness***. We already know that, if an equation can be proved from the axioms, it will be valid in all models; but we will also like to prove that, if every model of the theory satisfies a particular equation, it can actually be proved from the axioms of the theory.

Theorem 11 (Completeness for algebraic theories)**.** *Given $\mathbb{A}$ an algebraic theory, there exists a category $\mathcal{A}$ with a model $U \in \mathtt{Mod}_{\mathcal{A}}(\mathbb{A})$ such that, for every terms u, v,*

$$U \text{ satisfies } u = v \iff \mathbb{A} \text{ proves } u = v.$$

*This is called the **universal model** for $\mathbb{A}$. This theorem asserts that categorical semantics of algebraic theories are complete.*

Proof. Simply taking $\mathbb{A}$ with the identity functor, we have an universal model for $\mathbb{A}$. □

Note that this universal model needs not to be set-theoretic; but, even in this situation, we can always find a universal model in a presheaf category via the Yoneda embedding.

Proposition 18 (Yoneda embedding as a universal model)**.** *The Yoneda embedding $y \colon \mathbb{A} \to \widehat{\mathbb{A}}$ is a universal model for $\mathbb{A}$.*

Proof. It preserves finite products because it preserves all limits, hence it is a model. As it is a faithful functor, we know that any equation proved in the model is an eqation proved by the theory. □

17

CARTESIAN CLOSED CATEGORIES

17.1 EXPONENTIAL

Definition 77 (Exponential)**.** An **exponential** of A and B in a category with binary products is an object B^A with a morphism $e : B^A \times A \to B$ called *evaluation morphism*, such that, for any $f : C \times A \to B$ exists a unique $\widetilde{f} : C \to B^A$ such that the following diagram commutes

$$\begin{tikzcd}
B^A & B^A \times A \arrow[rd, "e"] & \\
C \arrow[u, dashed, "\exists! \widetilde{f}"] & C \times A \arrow[u, "\widetilde{f} \times id"] \arrow[r, "f"'] & B
\end{tikzcd}$$

An object A for which the exponentiation $-^A$ is always defined is called **exponentiable**.

Proposition 19 (Exponentials as adjoints)**.** *An object A in a category with binary products $\mathcal{C}$ is exponentiable if and only if the functor $(- \times A) : \mathcal{C} \to \mathcal{C}$ has a right adjoint.*

Proof. □

17.2 CARTESIAN CATEGORY

Definition 78 (Cartesian category)**.** A **cartesian category** is a category with all finite products.

Definition 79 (Cartesian closed category)**.** A **cartesian closed category** is a category with all finite products and exponentials.

The definition of cartesian closed category can be written in terms of existence of adjoints.

Proposition 20 (Cartesian closed categories and adjoints)**.** *Any category $\mathcal{C}$ is cartesian closed if and only if there exist right adjoints for the following functors*

- $! \colon \mathcal{C} \to 1$, *the unique functor to the terminal category;*
- $\Delta \colon \mathcal{C} \to \mathcal{C} \times \mathcal{C}$, *the diagonal functor;*
- $(- \times A) \colon \mathcal{C} \to \mathcal{C}$, *the product functor, for each* $A \in \mathcal{C}$.

Proof. We know that the product functor of an object has a right adjoint if and only if the object is exponentiable, as we saw in Proposition 19. □

Definition 80 (Category of small cartesian closed categories)**.** We call Ccc to the category of (small) cartesian closed categories with functors preserving finite products and exponentials as morphisms. These functors are called **cartesian closed functors**.

17.3 frames and locales

Proposition 21 (Completeness for posets)**.** *Complete posets are cocomplete. Cocomplete posets are complete.*

Proof. □

Definition 81 (Frames)**. Frames** are complete cartesian closed posets.

Definition 82 (Frame morphism)**.** A **frame morphism** is a function between frames preserving finite infima and arbitrary suprema.

18

HEYTING ALGEBRAS

In this section, we develop the notion of a **Heyting algebra** and show its differences with a Boolean algebra.

There is a correlation between classical propositional calculus and the Boolean algebra of the subsets of a given set. If we interpret a proposition p as a subset of a given universal set $P \subset U$ and fix an element $u \in U$, propositions can be translated to $u \in P$, logical connectives can be translated as

logic			subsets
$P \wedge Q$	and	intersection	$P \cap Q$
$P \vee Q$	or	union	$P \cup Q$
$\neg P$	not	complement	$\overline{P}$
$P \to Q$	implication	complement union	$\overline{P} \cup Q$

using crucially that $\neg P \wedge Q \equiv P \to Q$.

In the same way that Boolean algebras correspond to classical propositional logic, Heyting algebras correspond to intuitionistic propositional calculus. Its model on a set-like theory is not the subsets of a given set, but instead, only the *open* sets of a given topological space

logic			open sets
$P \wedge Q$	and	intersection	$P \cap Q$
$P \vee Q$	or	union	$P \cup Q$
$\neg P$	not	interior of the complement	$\text{int}\left(\overline{P}\right)$
$P \to Q$	implication	interior of complement and consequent	$\text{int}(\overline{P} \cup Q)$

where int is the topological interior of a set.

18.1 lattices and boolean algebras

Definition 83 (Lattice)**.** A **lattice** is a partially ordered set with all binary products and coproducts. It is a **bounded lattice** if it has all finite products and coproducts.

We will usually work with bounded lattices and simply call them *lattices*. A bounded lattice can be defined then by the following bidirectional inference rules

$$0 \leq x \leq 1 \qquad \frac{z \leq x \qquad z \leq y}{z \leq x \wedge y} \qquad \frac{x \leq z \qquad y \leq z}{x \vee y \leq z}$$

meaning that it has a terminal and a final object, in order to have all finte products and coproducts, and all binary products and coproducts.

Definition 84 (Lattice homomorphism)**.** A lattice homomorphism is a function between lattices preserving finite products and coproducts. That is, a function f such that

- $f(0) = 0, f(1) = 1,$
- $f(x \wedge y) = f(x) \wedge f(y),$
- $f(x \vee y) = f(x) \vee f(y);$

and the category of lattices and lattice homomorphisms is denoted by `Lat`.

A bounded lattice can also be defined as a set with $0, 1$ and two binary operations $\wedge, \vee$ satisfying

- $1 \wedge x = x$, and $0 \vee x = x$;
- $x \wedge x = x$, and $x \vee x = x$;
- $x \wedge (y \vee x) = x = (x \wedge y) \vee x$;
- $x \wedge y = y \wedge x$ and $x \vee y = y \vee x$.

This perspective allows us also to define a lattice object in any category as an object L with morphisms

$$\wedge \colon L \times L \to L, \quad \vee \colon L \times L \to L, \quad 0, 1 \colon \mathrm{I} \to L,$$

where I is the terminal object of the category; and commutative diagrams encoding the previous equations.

Definition 85 (Distributive lattice)**.** A **distributive lattice** is a lattice where

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z),$$

holds for all x, y, z.

Definition 86 (Complement)**.** A **complement** of a in a bounded lattice is an element $\overline{a}$ such that

$$a \wedge \overline{a} = 0, \qquad a \vee \overline{a} = 1.$$

Proposition 22 (The complement in distributive lattices is unique)**.** *If a complement of an element exists in a distributive lattice, it is unique.*

Proof. Given a with two complements x, y, we have that

$$x = x \wedge (a \vee y) = (x \wedge a) \vee (x \wedge y) = (y \wedge a) \vee (x \wedge y) = y \vee (x \wedge a) = y.$$

□

Definition 87 (Boolean algebra)**.** A **Boolean algebra** is a distributive bounded lattice in which every element has a complement.

Boolean algebras satisfy certain known properties such as the DeMorgan laws and the double negation elimination rule.

18.2 HEYTING ALGEBRAS

Definition 88 (Heyting algebra)**.** A **Heyting algebra**, also called **Brouwerian lattice**, is a bounded lattice which is cartesian closed as a category; that is, for every pair of elements x, y, the exponential y^x exists.

The exponential in Heyting algebras is usually written as $x \Rightarrow y$ and is characterized by its adjunction with the product

$$z \leq (x \Rightarrow y) \text{ if and only if } z \wedge x \leq y,$$

which can be expressed logically as

$$\frac{z \wedge x \leq y}{z \leq x \Rightarrow y}$$

Definition 89 (Heyting algebra homomorphism)**.** A **Heyting algebra homomorphism** is a lattice homomorphism between Heyting algebras which does preserve implication. The category of Heyting algebras is written as `Heyt`.

Proposition 23 (Boolean algebras are Heyting algebras)**.** *Every Boolean algebra is a Heyting algebra with exponentials given by*

$$(x \Rightarrow y) = \overline{x} \wedge y.$$

Proof. We will prove that

$$z \leq (\overline{x} \vee y) \text{ if and only if } z \wedge x \leq y.$$

If $z \leq (\overline{x} \vee y)$,

$$z \wedge x \leq (\overline{x} \vee y) \wedge x \leq (\overline{x} \wedge x) \vee (y \wedge x) \leq y \wedge x \leq y;$$

and if $z \wedge x \leq y$,

$$z = z \wedge 1 = z \wedge (\overline{x} \vee x) = (z \wedge \overline{x}) \vee (z \wedge x) \leq (z \wedge \overline{x}) \vee y \leq z \vee y.$$

□

Definition 90 (Negation)**.** The **negation** of x in a Heyting algebra is defined as

$$\neg x = (x \Rightarrow 0).$$

In general, we only have that $\neg\neg x \leq x$. An element for which $\neg\neg x = x$ is called a **regular** element.

Proposition 24. *In any Heyting algebra,*

1. $x \leq \neg\neg x$,
2. $x \leq y$ *implies* $\neg y \leq \neg x$,
3. $\neg x = \neg\neg\neg x$,
4. $\neg\neg(x \wedge y) = \neg\neg x \wedge \neg\neg y$,
5. $(x \Rightarrow x) = 1$,
6. $x \wedge (x \Rightarrow y) = x \wedge y$,
7. $y \wedge (x \Rightarrow y) = y$,
8. $x \Rightarrow (y \wedge z) = (x \Rightarrow y) \wedge (x \Rightarrow z)$.

Any bounded lattice L with an operation satisfying the last four properties is a Heyting algebra with this operation as implication.

Proof. We can prove the inequalities using the definition of implication.

1. By definition, $x \wedge (x \Rightarrow \bot) \leq \bot$.
2. Again, by definition, $\neg y \wedge x \leq \neg y \wedge y \leq \bot$.
3. Is a consequence of the first two inequalities.
4. We know that $x \wedge y \leq x, y$, and therefore $\neg\neg(x \wedge y) \leq \neg\neg x \wedge \neg\neg y$. We can prove $\neg\neg x \wedge \neg\neg y \leq \neg\neg(x \wedge y)$ using the definition of negation to get $\neg\neg x \wedge \neg\neg y \wedge \neg(x \wedge y) \leq \bot$, and then by reversing the definition of implication $\neg\neg y \wedge \neg(x \wedge y) \leq \neg\neg\neg x = x$. Applying the same reasoning to y, we finally get $x \wedge y \wedge \neg(x \wedge y) \leq \bot$.
5. Follows from $x \wedge 1 \leq x$.
6. Using the evaluation morphism, we know that $x \wedge (x \Rightarrow y) \leq y \leq x \wedge y$.
7. Using the definition of implication $y = y \wedge y \leq y \wedge (x \Rightarrow y)$.
8. The exponential $x \Rightarrow -$ is a right adjoint and it preserves products.

□

Proposition 25 (Complements are negations in Heyting algebras)**.** *If an element has a complement on a Heyting algebra, it must be* $\neg x$.

Proof. Let a a complement of x. By definition, $x \wedge a = \bot$ and therefore $a \leq \neg x$. The reverse inequality can be proven using the lattice properties as

$$\neg x = \neg x \wedge (x \vee a) = \neg x \wedge a.$$

□

Proposition 26 (Characterization of Boolean algebras). *A Heyting algebra is Boolean if and only if $\neg\neg x = x$ for every x; and if and only if $x \vee \neg x = 1$ for every x.*

Proof. In a Boolean algebra the complement is unique and $\neg\neg x = x$. Now, if $\neg\neg y = y$ for every y,

$$x \vee \neg x = \neg\neg(x \vee \neg x) = \neg(\neg x \wedge \neg\neg x) = \top;$$

and then, as $x \vee \neg x = \bot$, $\neg x$ must be the complement of x. We have used the fact that $\neg(x \vee y) = (\neg x) \wedge (\neg y)$ in any Heyting algebra. □

18.3 intuitionistic propositional calculus

Proposition 27 (Existence of free Heyting algebras).

Definition 91 (Intuitionistic propositional calculus). The **Intuitionistic Propositional Calculus** (IPC) is the free Heyting algebra over an infinite countable set $\{p_0, p_1, p_2, \dots\}$ of elements which are usually called *atomic propositions*.

- Classical propositional calculus
  Definition 92 (Classical propositional calculus). The **Classical Propositional Calculus** (CPC) is the free Boolean algebra over an infinite countable set $\{p_0, p_1, p_2, \dots\}$.

18.4 quantifiers as adjoints

Definition 93. Given a relation between sets $S \subseteq X \times Y$, the functors $\forall_p, \exists_p \colon \mathcal{P}(X \times Y) \to \mathcal{P}(Y)$ are defined as

- $\forall_p S = \{y \mid \forall x : \langle x, y\rangle \in S\}$, and
- $\exists_p S = \{y \mid \exists x : \langle x, y\rangle \in S\}$.

Theorem 12. *The functors $\exists_p, \forall_p$ are the left and right adjoints to the inverse image of the projection functor, $p^* \colon \mathcal{P}(Y) \to \mathcal{P}(X \times Y)$.*

Proof. □

Theorem 13. *Given any function on sets $f \colon Z \to Y$, the inverse image functor $f^* \colon \mathcal{P}Y \to \mathcal{P}Z$ has left and right adjoints, called $\exists_f$ and $\forall_f$.*

19

SIMPLY-TYPED λ-THEORIES

19.1 SIMPLY-TYPED λ-THEORIES

We will consider again the previously defined simply typed λ-calculus, but this time, as a theory instead of as a programming language. Recall that a *context* was a sequence of typed variables $x_1, x_2, \dots, x_n$, of the form

$$\Gamma = x_1 : A_1, \dots, x_n : A_n.$$

And any *typing judgment* was of the form $\Gamma \vdash a : A$. This time, the theory will also introduce equations of the form $\Gamma \vdash a = b : A$ as judgments.

In particular, we will work only with a fragment of system that we defined in "Extending the simply typed λ-calculus", namely, the fragment containing only arrow types, an unit type and product types.

Equality must now be defined over these typed terms. We have the following set of rules for it, clearly inspired by the usual properties of equivalence relations, β-reductions and η-reductions.

1. The assumptions needed to prove an equality can be weakened

$$\frac{\Gamma \vdash u = v : A}{\Gamma, w : B \vdash u = v : A}\ \textsc{weak}$$

2. Equality is reflexive, transitive and symmetric; that is, it is an equivalence relation

$$\frac{}{\Gamma \vdash u = u : A}\ \textsc{refl} \qquad \frac{\Gamma \vdash u = v : A}{\Gamma \vdash v = u : A}\ \textsc{symm} \qquad \frac{\Gamma \vdash u = v : A \qquad \Gamma \vdash v = w : A}{\Gamma \vdash u = w : A}\ \textsc{trans}$$

3. The unit type has only one element

$$\frac{}{\Gamma \vdash t = * : \top}$$

4. Reduction rules for products define the following equalities for pair constructors

$$\frac{\Gamma \vdash u = w : A \qquad \Gamma \vdash v = t : B}{\Gamma \vdash \langle u, v \rangle = \langle w, t \rangle : A \times B} \qquad \frac{}{\Gamma \vdash \texttt{fst}\langle u, v \rangle = u : A} \qquad \frac{}{\Gamma \vdash \texttt{snd}\langle u, v \rangle = v : B}$$

and the following ones for projections

$$\frac{\Gamma \vdash m = n : A \times B}{\Gamma \vdash \texttt{fst}\ m = \texttt{fst}\ n : A} \qquad \frac{\Gamma \vdash m = n : A \times B}{\Gamma \vdash \texttt{snd}\ m = \texttt{snd}\ n : B} \qquad \frac{}{\Gamma \vdash m = \langle \texttt{fst}\ m, \texttt{snd}\ m \rangle : A \times B}$$

5. Reduction rules for functions translate into the following equations, where the second one is directly derived from the β-rule

$$\frac{\Gamma \vdash f = g : A \to B \qquad \Gamma \vdash u = v : A}{\Gamma \vdash f\ u = g\ v : B} \qquad \frac{}{\Gamma \vdash (\lambda x.w)\ u = w[u/x] : A}$$

and the following ones, where the second one is directly derived from the η-rule

$$\frac{\Gamma, x : A \vdash w = t : B}{\Gamma \vdash (\lambda x.w) = (\lambda x.t) : A \to B} \qquad \frac{}{\Gamma \vdash (\lambda x.f\ x) = f : A \to B}$$

- Lambda theories In a λ-theory we may want to add a set of additional equations apart from those defined in the previous set of rules.
  Definition 94 (Simply typed lambda theory)**.** A simply-typed λ-theory is given by
  - a set of *basic types*;
  - a set of *basic constants* with their types; and
  - a set of *equations* on those terms, of the form $\Gamma \vdash a = b : A$, where Γ is the variable context.

19.2 interpretation of λ-theories

Definition 95 (Interpretation of a lambda theory)**.** An **interpretation** of a λ-calculus $\mathbb{T}$ in a cartesian closed category $\mathcal{C}$ is given by

1. an object $[\![A]\!] \in \mathbb{C}$ for every basic type A in $\mathbb{T}$.
2. a morphism $[\![c]\!] \colon 1 \to [\![A]\!]$ for every constant $c : A$.

The interpretation can be extended to all types using the terminal object, binary products and exponentials

- $[\![1]\!] = 1$;
- $[\![A \times B]\!] = [\![A]\!] \times [\![B]\!]$;
- $[\![A \Rightarrow B]\!] = [\![B]\!]^{[\![A]\!]}$.

Every context $\Gamma = x_1 : A_1, \dots, x_n : A_n$ can be interpreted as the object $[\![A_1]\!] \times \cdots \times [\![A_n]\!]$.

19.3 SYNTACTIC CATEGORIES

Definition 96 (Syntactic category)**.** The **syntactic category** of a λ-theory, $\mathcal{S}(\mathbb{T})$, is given by

- the types of the theory as objects; and
- the terms in context $x : A \vdash t : B$ as morphisms between A and B.

Two morphisms $x : A \vdash u : B$ and $x : A \vdash v : B$ are equal if $x : A \vdash u = v : B$ can be proved inside $\mathbb{T}$. The composition of two morphisms $a : A \vdash b : B$ and $b' : B \vdash c : C$ is given by $a : A \vdash c[b/b'] : C$.

Proposition 28 (Syntactic categories are cartesian closed)**.** *The syntactic category of a λ-theory is cartesian closed.*

Proof. We will prove that it has terminal, product and exponential objects. The terminal object is the unit type $\top$; it has a morphism $x : A \mid * : \top$ from any other type, and it is unique by virtue of the equality $\Gamma \vdash t = * : \top$.

Given two types A, B, its product is $A \times B$ with projections $m : A \times B \vdash \texttt{fst}\ m : A$ and $m : A \times B \vdash \texttt{snd}\ m : b$. Its universal property holds because for any other pair of morphisms $z : C \vdash a : A$ and $z : C \vdash b : B$, there exists a morphism

$$z : C \vdash \langle a, b \rangle : A \times B$$

satisfying that

- $z : C \vdash \texttt{fst}\ \langle a, b \rangle = a : A$,
- $z : C \vdash \texttt{snd}\ \langle a, b \rangle = b : B$,

and if any other morphism $z : C \vdash d : A \times B$ exists satisfying these same conditions, then

$$d = \langle \texttt{fst}\ d, \texttt{snd}\ d \rangle = \langle a, b \rangle.$$

Finally, given two types A, B, its exponential is $A \to B$ with the evaluation morphism

$$m : (A \to B) \times A \vdash (\texttt{fst}\ m)\ (\texttt{snd}\ m) : B.$$

Given any $p : C \times A \vdash q : B$, there exists a morphism

$$z : C \vdash \lambda x. q[\langle z, x \rangle / p] : A \to B$$

such that when we evaluate over any pair p we get

$$(\lambda x. q[\langle \texttt{fst}\ p, x \rangle / p])(\texttt{snd}\ p) = q[\langle \texttt{fst}\ p, \texttt{snd}\ p \rangle / p] = q[p/p] = q;$$

and if any other morphism $z : C \vdash d : A \to B$ exists satisfying this same condition, then $(d[\texttt{fst}\ p/z](\texttt{snd}\ p))$ would equal q and

$$\begin{aligned} \lambda x. q[\langle z, x \rangle / p] &= \lambda x. (d[\texttt{fst}\ p/z]\ (\texttt{snd}\ p))[\langle z, x \rangle / p] \\ &= \lambda x. (d[z/z]\ x) \\ &= d. \end{aligned}$$

□

Definition 97 (Model of a lambda-theory)**.** A **model** of a λ-theory $\mathbb{T}$ is a functor $M\colon \mathcal{S}(\mathbb{T}) \to \mathcal{C}$ preserving finite products and exponentials.

As we did with Lawvere theories, we can prove that categorical semantics for simply typed λ-theories is complete. By construction, given any theory $\mathbb{T}$,

$$\mathbb{T} \text{ proves } \Gamma \vdash x = y : A \quad\iff\quad \mathcal{S}(\mathbb{T}) \text{ satisfies } \Gamma \vdash x = y : A.$$

19.4 TRANSLATIONS, CATEGORY OF LAMBDA-THEORIES

Definition 98 (Translation)**.** A **translation** between λ-theories $\tau\colon \mathbb{T} \to \mathbb{U}$ is a model of $\mathbb{T}$ in the syntactic category $\mathcal{S}(U)$.

Definition 99 (Translation)**.** A **translation** between λ-theories $\tau\colon \mathbb{T} \to \mathbb{U}$ is given by

1. a type τA in $\mathbb{U}$ for each type A in $\mathbb{T}$; in such a way that

$$\tau 1 = 1, \qquad \tau(A \times B) = \tau A \times \tau B, \qquad \tau(A \to B) = \tau A \to \tau B.$$

2. a term $\tau c : \tau A$ in $\mathbb{T}$ for each term $c : A$ in $\mathbb{T}$; in such a way that

$$\begin{gathered}\tau(\mathtt{fst}\ t) = \mathtt{fst}\ (\tau t), \qquad \tau(\mathtt{snd}\ t) = \mathtt{snd}\ (\tau t), \quad \tau\langle u, v\rangle = \langle \tau u, \tau v\rangle, \\ \tau(tu) = (\tau t)(\tau u), \quad \tau(\lambda x : A.t) = \lambda x : \tau A.\tau t,\end{gathered}$$

A translation is also required to preserve all equations. That is, if $\Gamma \mid t = u : A$ can be proved in $\mathbb{T}$, $\tau\Gamma \mid \tau t = \tau u : \tau A$ should be provable in $\mathbb{U}$.

Definition 100 (Category of lambda-theories)**.** The category λ-$\mathtt{Thr}$ has λ-theories as objects and translations between them as morphisms.

Definition 101 (Isomorphism of types)**.** Two types in a λ-theory $\mathbb{T}$ are isomorphic if there exist terms $x : A \mid t : B$ and $y : B \mid u : A$ such that

$$x : A \mid u[t/y] = x : A, \quad \text{and} \quad y : B \mid t[u/x] = y : B,$$

are provable in $\mathbb{T}$.

Definition 102 (Equivalence of theories)**.** Two λ-theories $\mathbb{T}, \mathbb{U}$ are equivalent if there are a pair of translations $\tau : \mathbb{T} \to \mathbb{U}$ and $\sigma : \mathbb{U} \to \mathbb{T}$ such that

$$\sigma(\tau A) \cong A, \quad \text{and} \quad \tau(\sigma B) = B;$$

for any given types A, B.

19.5 internal language of a category

Definition 103 (Internal language)**.** The **internal language** of a small cartesian closed category $\mathcal{C}$ is a λ-theory given by

1. a *basic type* $\ulcorner A \urcorner$ for every object $A \in \mathcal{C}$;
2. a *constant* $\ulcorner f \urcorner\colon \ulcorner A \urcorner \to \ulcorner B \urcorner$ for every morphism $f : A \to B$;
3. the *identity* axiom
$$x : \ulcorner A \urcorner \mid \ulcorner \mathrm{id} \urcorner\, x = x : \ulcorner A \urcorner$$
for every A;
4. the *composition* axiom
$$x : \ulcorner A \urcorner \mid \ulcorner g \circ f \urcorner\, x = \ulcorner g \urcorner\, (\ulcorner f \urcorner\, x) : \ulcorner C \urcorner$$
for every pair of composable morphisms $f : A \to B$ and $g : B \to C$;
5. and *constants*
$$\begin{aligned} \mathsf{T} &: 1 \to \ulcorner 1 \urcorner \\ \mathsf{P}_{A,B} &: \ulcorner A \urcorner \times \ulcorner B \urcorner \to \ulcorner A \times B \urcorner \\ \mathsf{E}_{A,B} &: (\ulcorner A \urcorner \to \ulcorner B \urcorner) \to \ulcorner B^A \urcorner \end{aligned}$$
for any $A, B \in \mathcal{C}$.

Satisfying the following axioms

$$\begin{aligned} u : \ulcorner 1 \urcorner &\mid \mathtt{T}{*} = u : \ulcorner 1 \urcorner \\ z : \ulcorner A \times B \urcorner &\mid \mathtt{P}_{A,B}\langle \ulcorner \pi_0 \urcorner z, \ulcorner \pi_1 \urcorner z \rangle = z : \ulcorner A \times B \urcorner \\ w : \ulcorner A \urcorner \times \ulcorner B \urcorner &\mid \langle \ulcorner \pi_0 \urcorner (\mathtt{P}_{A,B} w), \ulcorner \pi_1 \urcorner (\mathtt{P}_{A,B} w) \rangle = w : \ulcorner A \urcorner \times \ulcorner B \urcorner \\ f : \ulcorner B^A \urcorner &\mid \mathtt{E}_{A,B}(\lambda x^{\ulcorner A \urcorner}. \ulcorner \mathtt{ev}_{A,B} \urcorner\, (\mathtt{P}_{A,B}\langle f, x \rangle)) = f : \ulcorner B^A \urcorner \\ g : \ulcorner A \urcorner \to \ulcorner B \urcorner &\mid \lambda x^{\ulcorner A \urcorner}. \ulcorner \mathtt{ev}_{\mathtt{A,B}} \urcorner (\mathtt{P}_{A,B}\langle \mathtt{E}_{A,B} g, x \rangle) = g : \ulcorner A \urcorner \to \ulcorner B \urcorner \end{aligned}$$

These axioms ensure the isomorphism between the terminal, product and exponential types of the category and the language; that is,

- $\ulcorner 1 \urcorner \cong 1$,
- $\ulcorner A \times B \urcorner \cong \ulcorner A \urcorner \times \ulcorner B \urcorner$,
- $\ulcorner B^A \urcorner \cong \ulcorner A \urcorner \to \ulcorner B \urcorner$.

20

LOCALLY CLOSED CARTESIAN CATEGORIES

In the same way that cartesian closed categories provide a categorical interpretation of the simply typed λ-calculus, locally closed cartesian categories will provide an interpretation of Martin-Löf type theories.

20.1 locally closed cartesian category

Definition 104 (The pullback functor)**.** Given a function $f\colon A \to B$ in any category $\mathcal{C}$ with all pullbacks, the **pullback functor** $f^*\colon \mathcal{C}/B \to \mathcal{C}/A$ is defined for any object $y\colon Y \to B$ as the object $f^*y\colon (f^*Y) \to A$ such that

$$\begin{tikzcd}
(f^*Y) \arrow[r] \arrow[d, "f^*y"'] & Y \arrow[d, "y"] \\
A \arrow[r, "f"] & B
\end{tikzcd}$$

is a pullback square. The functor is defined on any morphism $\alpha\colon y \to y'$ between two objects $y\colon Y \to B$, $y'\colon Y' \to B$ as the only morphism making the following diagram commute

$$\begin{tikzcd}
f^*Y \arrow[rrr] \arrow[rd, dashed, "\exists! f^*\alpha"] \arrow[rdd, bend right, "f^*y"'] & & & Y \arrow[ld, "\alpha"] \arrow[ldd, bend left, "y"] \\
& f^*Y' \arrow[r] \arrow[d, "f^*y'"'] & Y' \arrow[d, "y'"] & \\
& A \arrow[r, "f"'] & B &
\end{tikzcd}$$

where x and x' form pullback squares with y and y'. Note that the pullback functor is only defined up to isomorphism in objects and well-defined on morphisms by virtue of the universal property of pullbacks.

Proposition 29 (Left adjoint of the pullback functor)**.** *Any pullback functor $f^*\colon A \to B$ has a left adjoint Σ_f.*

Proof.

$$\hom(\Sigma_f x, y) \cong \hom(x, f^* y)$$

$$\begin{tikzcd}
X \arrow[dr, dashed] \arrow[drr, dashed, bend left] \arrow[ddr, bend right, "x"'] & & \\
& f^*Y \arrow[r] \arrow[d, "f^*y"] & Y \arrow[d, "y"] \\
& A \arrow[r] & B
\end{tikzcd}$$

□

Definition 105 (Right adjoint of the pullback functor)**.** A category $\mathcal{C}$ is locally cartesian closed if and only if the pullback functor $f^* \colon \mathcal{C}/B \to \mathcal{C}/A$ has a right adjoint, called $\Pi_f \colon \mathcal{C}/A \to \mathcal{C}/B$.

21

TOPOI

Follows [Leinster] and [Moerdijk]

21.1 SUBOBJECT CLASSIFIER

Definition 106 (Subobject classifier)**.** A **subobject classifier** is an object Ω with a monomorphism $\mathrm{true}\colon 1 \to \Omega$ such that, for every monomorphism $m : S \to X$, there exists a unique χ such that

$$\begin{tikzcd} S \arrow[r] \arrow[d, hook, "m"'] & 1 \arrow[d, "\mathrm{true}"] \\ X \arrow[r, dashed, "\chi"] & \Omega \end{tikzcd}$$

forms a pullback square.

The function χ is called the **characteristic function** of the monomorphism m.

Definition 107 (Subobjects)**.** A **subobject** of an object X is a monomorphism under the equivalence class given by isomorphism in the slice category $\mathcal{E}/X$.

Under this interpretation, each monomorphism represents a subobject, and two different monomorphisms $m : S \to X$ and $m' : S' \to X$ represent the same subobject if there exists an isomorphism $\varphi : m \cong m'$ in the slice category

$$\begin{tikzcd} S \arrow[rr, "\varphi"] \arrow[dr, "m"'] & & S' \arrow[dl, "m'"] \\ & X & \end{tikzcd}$$

For instance, subobjects in the category of sets correspond to subsets.

If we call $\mathrm{Sub}(X)$ to the class of subobjects of X, every map $f\colon X \to Y$ could induce a map $f^*\colon \mathrm{Sub}(Y) \to \mathrm{Sub}(X)$ given by pullback. For example, given any subobject represented by a monomorphism, $m : S' \to Y$, $f^*(m) : S \to X$ would be a monomorphism

$$\begin{tikzcd} S \arrow[r, dashed] \arrow[d, dashed, "f^*(m)"'] & S' \arrow[d, "m"] \\ X \arrow[r, "f"'] & Y \end{tikzcd}$$

such that this diagram is a pullback square.

When the base category $\mathcal{E}$ has all pullbacks and each $\mathsf{Sub}(X)$ is in fact a set, we can define the functor $\mathsf{Sub} : \mathcal{E}^{op} \to \mathsf{Set}$.

21.2 DEFINITION OF A TOPOS

Definition 108 (Topos)**.** An **elementary topos** (plural *topoi*) is a cartesian closed category with all finite limits and a subobject classifier.

Definition 109 (Logical morphism)**.** A **logical morphism** between two topoi, $F : \mathcal{E} \to \mathcal{E}'$ is a functor preserving finite limits, exponentials and the subobject classifier, up to isomorphism.

The category of finite sets and morphisms between them, FinSet,

21.3 ELEMENTARY THEORY OF THE CATEGORY OF SETS

In [Law64], Lawvere proposed an elementary theory of sets based on category theory by adjoining a set of axioms to the usual category theory axioms. In Set, the terminal object $\{*\}$ is a generator. That is, given two functions $f, g : X \to Y$, if $f \circ x = g \circ x$ for all $x : \{*\} \to X$, then $f = g$. In other words, if $f(x) = g(x)$ for all $x \in X$, then both functions must be the same.

Moreover, the terminal and the initial objects in Set are not isomorphic. A category where the terminal object is a generator and is not a zero object is a **well-pointed** category. A category is said to satisfy the **Axiom of choice** if every epimorphism has a right-side inverse. If we define sets using ZFC, the category Set satisfies the axiom of choice, that is, given any surjective function $f : A \to B$, there exists a function $k : B \to A$ such that $f \circ k = \mathrm{id}_B$.

Definition 110 (Category of sets)**.** A **category of sets** is a well-pointed topos with natural numbers satisfying the Axiom of Choice.

ZFC is stronger than ETCS: every provable theorem in ETCS is provable in ZFC, but the converse is not true. In particular, ETCS is equivalent to a fragment of ZFC, called *restricted ZFC*. This relation is detailed in [MM94].

Part V

TYPE THEORY

This section is mainly based on [Uni13] and [Course on Hott, Harper].

22

MARTIN-LÖF TYPE THEORY

Type theory can be seen as an intuitionistic foundation of mathematics.

- **Martin-Löf Type Theory** (MLTT) can be used as a constructive and computational foundation of mathematics. It is not directly based on first-order predicate logic, but only interpreted in it via the Curry-Howard isomorphism. It comes in two flavours, intensional and extensional.
- **Intensional type theory** (ITT) is an intuitionistic type theory serving as the core to many other type theories.
- **Extensional type theory** (ETT) extends Intensional Type Theory with equality of reflection and uniqueness of identity proofs. Types behave like sets when we interpret them in this setting, and therefore, it can be seen as an intuitionistic theory of sets.

The principal difference between types and sets is that In Type theory, membership of an element to a type, as in $a : A$, is a *judgement* of the theory instead of a *proposition*. In practice, this means that that we can write things like $\forall a.a \in A \to (a,a) \in A \times A$, as $a \in A$ **is** a proposition; but we cannot write $\prod_{a:A}(a,a) : A \times A$. A type declaration is not part of the language.

This section follows [Course on Hott, Harper] and [Hott].

We introduce ITT with Naturals, Sigma, Pi, Identity types and Universes. [Martin-Löf 73]

22.1 CONTEXTS AND JUDGMENTS

As we did when we described simply typed λ-calculus; we will present a type theory using contexts, substitutions and some inference rules. We consider three different kinds of judgments for this formal system, namely,

- Γ `ctx`, expressing that Γ is a context;
- $\Gamma \vdash a : A$, expressing that, from the context Γ, it follows that a is of type A; and

- $\Gamma \vdash a \equiv a' : A$, expressing that, from the context Γ, it follows that a is *definitionally equal* to a'. It is important not to confuse this notion of equality with the notion of *propositional equality* we will describe later.

Contexts, in particular, will be given by a (possibly empty) list of type declarations

$$x_1{:}A_1,\ x_2{:}A_2,\ \dots,\ x_n{:}A_n,$$

and we will consider as valid the judgement saying that the empty context is a context ($\cdot$ `ctx`), and the judgement saying that, given a context $x_1{:}A_1,\ x_2{:}A_2,\ \dots, x_{n-1}{:}A_{n-1}$ and any type A_n, we can take a new unused variable to form a new context

$$x_1{:}A_1,\ x_2{:}A_2,\ \dots, x_{n-1}{:}A_{n-1}, x_n{:}A_n\ \texttt{ctx}.$$

22.2 TYPE UNIVERSES

If we want to blur the difference between types and terms, we will need type declarations for types. Types whose elements are types are called ***universes***. We would like, then, to define a single universe of all types $\mathcal{U}$ containing even itself as an element, $\mathcal{U} : \mathcal{U}$; but this leads to paradoxes. In particular, we can encode a particular version of Russell's paradox.

To avoid having to deal with paradoxes, we will postulate a ***hierarchy of cumulative universes***; that is, we will consider a list of inclusions between countably many type universes

$$\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \dots$$

and such that $a : \mathcal{U}_i$ implies $a : \mathcal{U}_{i+1}$. The relevant inference rules are

$$\frac{}{\Gamma \vdash \mathcal{U}_i : \mathcal{U}_{i+1}} \qquad \frac{\Gamma \vdash A : \mathcal{U}_i}{\Gamma \vdash A : \mathcal{U}_{i+1}}$$

for each i in the natural numbers.

Universe indices, however, are usually implicitly assumed when writing ITT; as we will mostly work with types from some fixed universe of the hierarchy $\mathcal{U}$ and only refer to higher universes when necessary. This should not lead to confusion: a derivation will be only valid if we can assign indices to the universes consistently, even if we do not explicitly write them.

22.3 DEFINING TYPES

Types will be defined by the formation and elimination rules we can apply to them. We will follow a general pattern for specifying the typing rules of ITT. They will be classified into

- **formation rules**, indicating how to create new types of a particular kind;
- **introduction rules**, indicating how to create new terms of a particular type;
- **elimination rules**, indicating how to use elements of a particular type;
- **β-reductions**, indicating how elimination acts on terms;
- **η-reductions**, defining some kind uniqueness principle.

Depending in how the uniqueness principle is defined, we classify types into negative and positive types.

- Uniqueness of **negative types** states that every element of the type can be reconstructed by applying eliminators to it and then applying a constructor. Products are an example of negative types.
- Uniqueness of **positive types** states that every funtion from the type is determined by some data. Coproducts are an example of positive types.

22.4 DEPENDENT FUNCTION TYPES

Dependent function types, or **Π-types**, are the generalized version of the function types in simply-typed lambda calculus. The elements of $\prod_{x:A} B(x)$ are functions with the type A as domain and a changing codomain $B(x)$, depending on the specific element to which the function is applied. This type is often written also as $\Pi(x : A), B(x)$ to resemble the universal quantifier; under the *propositions as types* interpretation, it would correspond to the proof that a proposition B holds for any $x : A$, that is, $\forall(x : A), B(x)$.

Definition 111 (Dependent function type)**.** The following rules apply for the dependent function type:

- its formation rule sinthetizes the fact that the codomain type can depend on the argument to which the function is applied,
$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma, x : A \vdash B : \mathcal{U}_i}{\Gamma \vdash \prod_{x:A} B : \mathcal{U}_i}\ \Pi\text{-FORM}$$
- its introduction rule its a generalized version of λ-abstraction,
$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x.b : \prod_{x:A} B}\ \Pi\text{-INTRO}$$
- and its elimination rule generalizes function application,
$$\frac{\Gamma \vdash f : \prod_{x:A} B \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B[a/x]}\ \Pi\text{-ELIM}$$
- its β-rule is similar to simply typed lambda calculus' β-rule, with the difference that it has to specify a substitution on the type of the codomain
$$\frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda x.b)\ a \equiv b[a/x] : B[a/x]}\ \Pi\text{-COMP}$$

- and its η-rule is also similar to simply typed lambda calculus' one

$$\frac{\Gamma \vdash f : \prod_{x:A} B}{\Gamma \vdash f \equiv \lambda x.f(x) : \prod_{x:A} B}\ \Pi\text{-UNIQ}$$

In the particular case in which x does not appear in B so that B is constant and does not depend on A, we obtain our usual **function type**. In ITT, we define the function type as a particular case of the dependent function type, $A \to B :\equiv \prod_{x:A} B$.

Polymorphic functions can be defined in terms of dependent function types and universes. We can see a polymorphic term as a dependent function taking a type as its first argument. For example, the identity function id : $\prod_{A:\mathcal{U}} A \to A$ can be defined as id $:\equiv \lambda(A : \mathcal{U}).\lambda(x : A).x$. When applied, the type should be passed as an argument, but it is a common convention to omit arguments when they are obvious from the context and only refer to them when typechecking. A more complex example of polymorphic function is *function composition*, which is also an example of higher-order function. Its type signature takes three types as arguments and two functions between these types,

$$\circ : \prod_{A:\mathcal{U}_i} \prod_{B:\mathcal{U}_i} \prod_{C:\mathcal{U}_i} (A \to B) \to (B \to C) \to A \to C$$

and it is defined as

$$\circ :\equiv \lambda(A : \mathcal{U}_i).\lambda(B : \mathcal{U}_i).\lambda(C : \mathcal{U}_i).\ \lambda g.\lambda f.\lambda x.\ g(f(x)).$$

However, we face a notational nuisance here: to explicitly write down all the types each time we want to notate a function would be very wordy. Instead, we will allow arguments to be determined implicitely. Implicit arguments must be inferred by the context and its supression is not formalized as part of our type theory. At the same time, we will use infix notation whenever it is easier to read.

Thus, we will write $f \circ g$ instead of $\circ(A, B, C, f, g)$.

22.5 DEPENDENT PAIR TYPES

Dependent pair types, or **Σ-types**, can be seen as a generalized version of the product type, but they can be also particularized in the union type. The elements of $\sum_{x:A} B(x)$ are pairs where the first element is of type A and the second element is of type $B(x)$, where x is the first one; that is, the type of the second component depends on the first component. This type is often written as $\Sigma(x : A), B(x)$ and it corresponds to the intuitionistic existential quantifier under the *propositions as types* interpretation. That is, the proof of $\exists(x : A), B(x)$ must be seen as a pair given by an element x and a proof of $B(x)$.

Definition 112 (Dependent pair type)**.** The following rules apply for the dependent pair type:

- its formation rule is similar to that of the product

$$\frac{\Gamma \vdash A \colon \mathcal{U}_i \qquad \Gamma, x : A \vdash B \colon \mathcal{U}_i}{\Gamma \vdash \sum_{x:A} B : \mathcal{U}_i}\ \Sigma\text{-FORM}$$

- a pair can be constructed by its two components

$$\frac{\Gamma, x : A \vdash B : \mathcal{U}_i \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : B[a/x]}{\Gamma \vdash (a,b) : \sum_{x:A} B}\ \Sigma\text{-INTRO}$$

- its elimination rule tell us that we can use a pair using its two elements, x, y, to create a new one

$$\frac{\Gamma, z : \sum_{x:A} B \vdash C : \mathcal{U}_i \qquad \Gamma, x : A, y : B \vdash g : C[(x,y)/z] \qquad \Gamma \vdash p : \sum_{x:A} B}{\Gamma \vdash \mathrm{ind}_{\sum}([z].C, [x].[y].g, p) : C[p/z]}\ \Sigma\text{-ELIM}$$

- and its β-rule substitutes the two elements of the pair onto an element

$$\frac{\Gamma, z : \sum_{x:A} B \vdash C : \mathcal{U}_i \qquad \Gamma, x : A, y : B \vdash g : C[(x,y)/z] \qquad \begin{array}{c}\Gamma \vdash a : A \\ \Gamma \vdash b : B[a/x]\end{array}}{\Gamma \vdash \mathrm{ind}_{\sum}([z].C, [x].[y].g, (a,b)) \equiv g[a/x][b/y] : C[(a,b)/z]}\ \Sigma\text{-COMP}$$

Note that we do not postulate an η-rule for the dependent pair type. We will show later that it follows from these rules.

The elimination rule says that we can define a function from the dependent pair type $\prod_{x:A} B$ to an arbitrary type C by assuming some $x : A$ and $y : B(x)$ and constructing a term of type C with them.

For instance, we can declare of the first projection to be $\mathtt{pr}_1 : (\sum_{x:A} B(x)) \to A$ and define it as $\mathtt{pr}_1((a,b)) :\equiv a$. The second projection is slightly more complex, as it must be a dependent function

$$\mathtt{pr}_2 : \prod_{p:\sum_{x:A} B(x)} B(\mathtt{pr}_1(p))$$

whose type depends on the first projection. It can be again defined as $\mathtt{pr}_2((a,b)) :\equiv b$.

An interesting example arises when we consider a function that builds a function from a term R that can be regarded as a proof-relevant binary relation, in that any term of type $R(x,y)$ is a witness of the relation between some particular x and y.

$$\mathtt{ac} : \left(\prod_{x:A}\sum_{y:B} R(x,y)\right) \to \left(\sum_{f:A\to B}\prod_{x:A} R(x, f(x))\right)$$

Under the propositions as types interpretation, this type represents the **axiom of choice**; that is, if for all $x \in A$ there exists a $y \in B$ such that $R(x,y)$, then there exists a function $f : A \to B$ such that $R(x, f(x))$ for all $x \in A$.

The actual **axiom of choice** can be shown to be independent of Zermelo-Fraenkel set theory. However, maybe surprisingly, this type theoretic version can be directly proved from the axioms of our system. The function $\mathtt{ac}$ can be constructed as

$$\mathtt{ac} :\equiv \lambda g.\, (\mathtt{pr}_1 \circ g, \mathtt{pr}_2 \circ g)\,,$$

and we can check that, in fact, if $g : \prod_{x:A} \sum_{y:B} R(x,y)$, then

- $\mathtt{pr}_1 \circ g$ is of type $A \to B$, and
- $\mathtt{pr}_2 \circ g$ is of type $\prod_{x:A} R(x, \mathtt{pr}_1(g(x)))$;

effectively proving that $\mathtt{ac}(g)$ has type $\sum_{f:A\to B} \prod_{x:A} R(x, f(x))$.

Why the type-theoretic version of the axiom of choice is, instead, a **theorem of choice**? The crucial notion here is that no choice is actually involved. The dependent pair Σ differs from the classical existential quantification $\exists$ in that it is constructive, that is, any witness of $\sum_{a:A} B$ has to actually provide an element of A such that $B(a)$; while a proof of $\exists a \in A, B$ does not need to point to any particular element of A.

Algebraic structures can be also described as dependent pairs. For instance, we can define a **magma** to be a type A endowed with a binary operation $A \to A \to A$. This can be encoded in a type of magmas as

$$\mathtt{Magma} :\equiv \sum_{A:\mathcal{U}} (A \to A \to A).$$

After we introduce identity types, we will be able to add constraints to our structures that will allow us to define more complex algebraic structures such as groups or categories.

22.6 unit, empty, coproduct and boolean types

The **empty** type, $0 \colon \mathcal{U}$, has no inhabitants and a function $f \colon 0 \to C$ is defined without having to give any equation.

Definition 113 (Empty type)**.** The following rules apply for the empty type:

- it can be formed without any assumption,
$$\frac{}{\Gamma \vdash 0 : \mathcal{U}_i}\ 0\text{-FORM}$$
- and it can be used without any restriction,
$$\frac{\Gamma, x : 0 \vdash C : \mathcal{U}_i \qquad \Gamma \vdash a : 0}{\Gamma \vdash \mathrm{ind}_0([x].C, a) : C[a/x]}\ 0\text{-ELIM}$$

Note that the empty type has no introduction and β-rules. We cannot compute with the empty type and it should not be possible to create an element of that type.

The **unit** type, $1 : \mathcal{U}$ has only one inhabitant. To define a function $f : 1 \to C$ amounts to choose on element of type C.

Definition 114 (Unit type)**.** The following rules apply for the unit type:

- it can be formed without any assumption
$$\frac{}{\Gamma \vdash 1 : \mathcal{U}_i}\ 1\text{-FORM}$$
- and its only instance can be also introduced without any assumption,
$$\frac{}{\Gamma \vdash \star : 1}\ 1\text{-INTRO}$$

- it can be used to choose an instance of any type

$$\frac{\Gamma, x : 1 \vdash C : \mathcal{U}_i \qquad \Gamma \vdash c : C[\star/x] \qquad \Gamma \vdash a : 1}{\Gamma \vdash \mathrm{ind}_1([x].C, c, a) : C[a/x]}\ \textsc{1-elim}$$

- and its beta rule computes the element we had chosen

$$\frac{\Gamma, x : 1 \vdash C : \mathcal{U}_i \qquad \Gamma \vdash c : C[\star/x]}{\Gamma \vdash \mathrm{ind}_1([x].C, c, \star) \equiv c : C[\star/x]}\ \textsc{1-comp}$$

Uniqueness need not to be postulated, it can be proved using these axioms.

Definition 115 (Coproduct type)**.** The following rules apply for coproduct types:

- there is a coproduct type for any two types,

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash B : \mathcal{U}_i}{\Gamma \vdash A + B : \mathcal{U}_i}\ \textsc{+-form}$$

- a term can be introduced in two different ways

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash B : \mathcal{U}_i \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathtt{inl}(a) : A + B}\ \textsc{+-intro1}$$

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash B : \mathcal{U}_i \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathtt{inr}(b) : A + B}\ \textsc{+-intro2}$$

- and its elimination rule has to consider both cases

$$\frac{\begin{array}{c} \Gamma, x{:}A \vdash c : C[\mathtt{inl}(x)/z] \\ \Gamma, z{:}(A+B) \vdash C : \mathcal{U}_i \qquad \Gamma, y{:}B \vdash d : C[\mathtt{inr}(y)/z] \qquad \Gamma \vdash e : A + B \end{array}}{\Gamma \vdash \mathtt{ind}_{A+B}([z].C, [x].c, [y].d, e) : C[e/z]}\ \textsc{+-elim}$$

- in particular, we have two β-rules, each one for each different way in which a term can be introduced

$$\frac{\begin{array}{c} \Gamma, x{:}A \vdash c : C[\mathtt{inl}(x)/z] \\ \Gamma, z{:}(A+B) \vdash C : \mathcal{U}_i \qquad \Gamma, y{:}B \vdash d : C[\mathtt{inr}(y)/z] \qquad \Gamma \vdash a : A \end{array}}{\Gamma \vdash \mathtt{ind}_{A+B}([z].C, [x].c, [y].d, \mathtt{inl}(a)) \equiv c[a/x] : C[\mathtt{inl}(a)/z]}\ \textsc{+-comp1}$$

$$\frac{\begin{array}{c} \Gamma, x{:}A \vdash c : C[\mathtt{inl}(x)/z] \\ \Gamma, z{:}(A+B) \vdash C : \mathcal{U}_i \qquad \Gamma, y{:}B \vdash d : C[\mathtt{inr}(y)/z] \qquad \Gamma \vdash b : B \end{array}}{\Gamma \vdash \mathtt{ind}_{A+B}([z].C, [x].c, [y].d, \mathtt{inr}(b)) \equiv d[b/x] : C[\mathtt{inr}(b)/z]}\ \textsc{+-comp2}$$

22.7 natural numbers

Although we will see later that it can be seen as a particular case of inductive datatypes, a **natural numbers** type can be directly defined in the core of our type theory. Its introduction rules will match Peano's axioms and its elimination and β-rules will provide us with the notion of induction.

Definition 116 (Natural numbers)**.** The following rules apply for natural numbers:

- the natural numbers type can be formed without any assumption,

$$\frac{}{\Gamma \vdash \mathbb{N} : \mathcal{U}_i}\ \mathbb{N}\text{-FORM}$$

- a natural number can be introduced in two ways, as zero or as the successor of a natural number

$$\frac{}{\Gamma \vdash 0 : \mathbb{N}}\ \mathbb{N}\text{-INTRO1} \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathtt{succ}(n) : \mathbb{N}}\ \mathbb{N}\text{-INTRO2}$$

- we can apply induction on natural numbers

$$\frac{\begin{array}{c}\Gamma \vdash c_0 : C[0/x] \\ \Gamma, x{:}\mathbb{N} \vdash C : \mathcal{U}_i \qquad \Gamma, x{:}\mathbb{N}, y{:}C \vdash c_s : C[\mathtt{succ}(x)/x] \qquad \Gamma \vdash n : \mathbb{N}\end{array}}{\Gamma \vdash \mathtt{ind}_{\mathbb{N}}([x].C, c_0, [x].[y].c_s, n) : C[n/x]}\ \mathbb{N}\text{-ELIM}$$

- and it must be interpreted recursively for zero or a successor with the following two β-rules

$$\frac{\begin{array}{c}\Gamma \vdash c_0 : C[0/x] \\ \Gamma, x{:}\mathbb{N} \vdash C : \mathcal{U}_i \qquad \Gamma, x{:}\mathbb{N}, y{:}C \vdash c_s : C[\mathtt{succ}(x)/x]\end{array}}{\Gamma \vdash \mathtt{ind}_{\mathbb{N}}([x].C, c_0, [x].[y].c_s, 0) \equiv c_0 : C[0/x]}\ \mathbb{N}\text{-COMP1}$$

$$\frac{\begin{array}{c}\Gamma \vdash c_0 : C[0/x] \\ \Gamma, x{:}\mathbb{N} \vdash C : \mathcal{U}_i \qquad \Gamma, x{:}\mathbb{N}, y{:}C \vdash c_s : C[\mathtt{succ}(x)/x]\end{array}}{\Gamma \vdash \mathtt{ind}_{\mathbb{N}}([x].C, c_0, [x].[y].c_s, \mathtt{succ}\ n) \equiv}\ \mathbb{N}\text{-COMP2}$$
$$c_s[n/x][\mathrm{ind}_{\mathbb{N}}([x].C, c_0, [x].[y].c_s, n)/y] : C[\mathtt{succ}(n)/x]$$

Any function on natural numbers can be defined in two equivalent ways, either using the induction principle or its defining equations.

- If we use the induction principle, a function from the naturals to the type C must be defined as

  $$f :\equiv \mathtt{ind}_{\mathbb{N}}(C, c_0, c_s)$$

  where c_s can depend on an element of type C. Two defining equations capturing the definition could be written in this case as

  $$\begin{aligned} f(0) &:\equiv c_0, \\ f(\mathtt{succ}(n)) &:\equiv c_s, \end{aligned}$$

  where c_s would depend on an element of type C that can be interpreted as being $f(n)$, that is, a recursive call to the function.
- Conversely, we could define a function by its two defining equations. If we had

  $$\begin{aligned} f(0) &:\equiv \Phi_0, \\ f(\mathtt{succ}(n)) &:\equiv \Phi_s, \end{aligned}$$

  where $\Phi_s : C$ depends on $f(n)$, we could substitute $f(n)$ by a variable r and equivalently write the definition as

  $$f :\equiv \mathtt{ind}_{\mathbb{N}}(C, \Phi_0, \lambda n.\lambda r.\Phi_s).$$

This can be done in general with types whose induction principle can be split in multiple cases. We will call **pattern matching** to this style of defining functions and it will be used from now on because of its simplicity. However, we must be careful when doing this, as it could create the false impression that arbitrary recursion is allowed in our definitions. An equation such as $f(\mathtt{succ}(n)) :\equiv f(\mathtt{succ}(\mathtt{succ}(n)))$, for example, is not valid.

Our first example will be the definition of addition on natural numbers. It will be an infix function $+ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ given by

$$\begin{aligned} 0 + m &:\equiv m, \\ (\mathtt{succ}\ n) + m &:\equiv \mathtt{succ}(n+m), \end{aligned}$$

where we are applying the induction principle to the first argument. In other words,

$$+ :\equiv \lambda(n{:}\mathbb{N}).\lambda(m{:}\mathbb{N}).\mathtt{ind}_{\mathbb{N}}(\mathbb{N}, m, [s].[r].\mathtt{succ}\ r, n).$$

Under this definition it can be shown that, for example, $2+2 :\equiv 4$. However, if we wanted to prove the fact that addition is commutative, we would have no way of expressing this in terms of an extensional equality. We would need a notion of propositional equality, that is, an equality $=$ that could be used as a type in an expression, in order to write something like

$$\mathtt{comm}_{+} : \prod_{n:\mathbb{N}} \prod_{m:\mathbb{N}} (n+m = m+n).$$

22.8 identity types

Under a *propositions as types* interpretation, the proposition that two elements of a type are equal, should correspond to a type; and therefore, equality should correspond to a family of types,

$$= : \prod_{A:\mathcal{U}} A \to A \to \mathcal{U}.$$

We will write the equality type between $a, b : A$ interchangeably as $a =_A b$ or $\mathrm{Id}_A(a,b)$.

Definition 117 (Identity types)**.** The following rules apply for identity types:

- an identity type is defined between any two elements of the same type

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Gamma \vdash a =_A b : \mathcal{U}_i}\ =\text{-FORM}$$

- the only way to introduce an identity is using the principle of **reflexivity**, that is, there exists an element `refl` of type $a =_A a$ for each $a : A$,

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathrm{refl} : a =_A a}\ =\text{-INTRO}$$

- its elimination rule of $p : x =_A y$ allows us to create an instance of a particular type, or prove a proposition, simply assuming that x and y are in fact the same element and $p = \mathrm{refl}_z$,

$$\frac{\Gamma, x{:}A, y{:}A, p{:}(x =_A y) \vdash C : \mathcal{U}_i \qquad \Gamma \vdash a : A \qquad \Gamma, z{:}A \vdash c : C[z/x][z/y][\mathrm{refl}/p] \qquad \Gamma \vdash b : A \qquad \Gamma \vdash q : a =_A b}{\Gamma \vdash \mathtt{ind}_{=_A}([x].[y].[p].C, [z].c, a, b, q) : C[a, b, q/x, y, p]} \; = \text{-ELIM}$$

- and its β-rule simply substitutes two equal instances by the same one

$$\frac{\Gamma, x{:}A, y{:}A, p{:}x =_A y \vdash C : \mathcal{U}_i \qquad \Gamma, z{:}A \vdash c : C[z, z, \mathtt{refl}_z/x, y, p] \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathtt{ind}_{=_A}([x].[y].[p].C, [z].c, a, a, \mathtt{refl}_a) \equiv c[a/z] : C[a/x][a/y][\mathtt{refl}_a/p]} \; = \text{-COMP}$$

Note that the introduction rule on types is a dependent function called **reflexivity** that provides us with a basic way of constructing an element of type $(a =_A b) : \mathcal{U}$,

$$\mathtt{refl} : \prod_{a:A} (a =_A a).$$

In particular, any two definitionally equal terms, such as $2 + 2 :\equiv 4$, are also propositionally equal. In fact, $\mathtt{refl}_4 : 2 + 2 = 4$ is a well-typed statement, precisely because of the fact that $2 + 2$ and 4 are definitionally equal.

The elimination principle and the β-rule for identity types are usually called the **path induction** principle. It states that, given a family of types parametrized over the identity type,

$$C : \prod_{x,y:A} (x =_A y) \to \mathcal{U},$$

every function defined over the refl element, $c : \prod_{z:A} C(z, z, \mathtt{refl}_z)$, can be extended to any equality term as

$$f : \prod_{x:A} \prod_{y:A} \prod_{p:x=y} C(x, y, p),$$

and it follows from the β-rule that $f(x, x, \mathtt{refl}_x) :\equiv c(x)$, that is, that it is in fact an extension.

This **path induction** principle could be wrapped into a function

$$\mathtt{ind}_= : \prod_{\left(C : \prod_{x:A} \prod_{y:A} (x=y) \to \mathcal{U}\right)} \left(\prod_{x:A} C(x, x, \mathtt{refl_x}) \to \prod_{x:A} \prod_{y:A} \prod_{p:x=y} C(x, y, p) \right)$$

and the β-rule could be replaced by $\mathtt{ind}_=(C, c, x, x, \mathtt{refl}_x) :\equiv c(x)$. We will use path induction to prove a the principle of **indiscernability of identicals**, that is, the fact that, for every family of types $C : A \to \mathcal{U}$, there is a function between the type associated to propositionally equal elements, that is,

$$f : \prod_{x:A} \prod_{y:A} \prod_{p:x=y} C(x) \to C(y),$$

such that, when applied to reflexivity, it outputs the identity function, $f(x, x, \mathtt{refl}_x) :\equiv \mathtt{id}_{C(x)}$.

We can prove this principle for any $C : A \to \mathcal{U}$ by defining

$$f :\equiv \mathtt{ind}_{=} \left(\lambda x. \lambda y. \lambda p. C(x) \to C(y), \mathtt{id}_{C(x)} \right),$$

and the fact that $f(x, x, \mathtt{refl}_x) :\equiv \mathtt{id}_{C(x)}$ follows from the β-rule.

22.9 INDUCTIVE TYPES

Inductive types provide the intuitive notion of a free type generated by a finite set of constructors. That is, the only elements of an inductive type should be those that can be obtained by applying a fixed finite set of constructors. Thus, we can define functions over inductive types using certain induction principles that define a specific action for each one of the possible constructors.

The **W-type** (or *Brower ordinal*) $\mathsf{W}_{a:A} B(a)$, also written as $\mathsf{W}(a : A), B(a)$, can be thought as an inductive type whose set of constructors is indexed by the type A, such that the constructor indexed by the particular element a has arguments indexed by $B(a)$. In other words, it is a well-founded tree where every node is labeled by an element of a and has a set of child nodes indexed by the type $B(a)$.

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma, x : A \vdash B : \mathcal{U}}{\Gamma \vdash \mathsf{W}_{x:A} B : \mathcal{U}} \text{ W-FORM}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x{:}B(a) \vdash \mathsf{W}_{x:A} B}{\Gamma \vdash \mathtt{sup}[a]([x].w) : \mathsf{W}_{x:A} B} \text{ W-INTRO}$$

$$\frac{\begin{array}{c}\Gamma, z : \mathsf{W}_{x:A} B \vdash P : \mathcal{U} \\ \Gamma, a{:}A, p : B(a) \to \mathsf{W}_{x:A} B, h : \prod_{b:B(a)} P(p(b)) \vdash M : P(\mathtt{sup}(p))\end{array}}{\Gamma, z{:} \mathsf{W}_{x:A} B \vdash \mathtt{ind}_{\mathsf{W}}(a, p, [h].M) : P(z)} \text{ W-ELIM}$$

The relevant endofunctor must be polynomial.

Not all endofunctors have initial algebras, but all polynomial functors do have initial algebras.

Each W-type determines a functor on types given by

$$F(X) = \sum_{a:A} B(a) \to X.$$

Functors of this form are called **polynomial functors**, as they can be written as $\sum_{a:A} X^{B(a)}$ and thought as a generalized polynomial function.

In fact, this functor is an F-algebra with the function

$$\lambda(a, w).\mathrm{sup}[a](w) : F(X) \to X.$$

23

PROGRAMMING IN MARTIN-LÖF TYPE THEORY

Martin-Löf type theory can also be regarded as a programming language in which is possible to formally specify and prove theorems about the code itself. It is similar to the Calculus of Constructions that we mentioned earlier.

Formal proofs in MLTT can be written in this programming language and checked by a machine. Some programming languages offering this complex type system are

- **Agda** implements a variant of MLTT and can be used as a programming language or a proof assistant;
- **Coq** [04] was developed in 1984 in INRIA; it implements Calculus of Constructions and was used, for example, to check a proof of the Four Color Theorem;
- **Idris** implements a different version of identity types than that in MLTT and aims to be a useful programming language instead of a proof assistant;
- **NuPRL** [CAB+86] implements Extensional Type Theory;
- **Cubical** [CCHM16] provides an implementation of Cubical Type Theory;
- **RedPRL** is still in beta and provides an implementation of Cubical Type Theory in the NuPRL style.

In this text, we will use Agda as our proof assistant and we will mechanize proofs in Martin-Löf type theory using it. We will introduce Agda using ideas from [McB17].

23.1 MARTIN-LÖF TYPE THEORY IN AGDA

In this chapter, we will describe how to program in MLTT using Agda as proof assistant. An Agda file consists of type declarations and definitions stated with =, as in the following example.

```
- An identity function
id : ∀{l} {A : Set l} → A → A
id a = a

- A constant function
const : ∀{l} {A B : Set l} → A → B → A
const a b = a

- Composition of functions
_∘_ : ∀{l} {A B C : Set l} → (B → C) → (A → B) → A → C
(g ∘ f) x = g (f x)
```

This code exemplifies multiple idiosyncratic features of Agda when compared to our previous presentation of MLTT.

- The type of types, $\mathcal{U}$, is written as `Set`; however, it has nothing to do with our usual *set-theoretical sets*. The hierarchy of universes in Agda is written as

  $$\mathtt{Set} : \mathtt{Set}_1 : \mathtt{Set}_2 : \dots$$

  We will see later that it is possible to use compiler flags to postulate the collapse of this hierarchy into a single `Set : Set` and derive a contradiction from this assumption.
  If we want a definition to work at all levels of the universe hierarchy, we need to explicitly write `∀{l}...Set l` instead of simply write `Set`.
- Dependent function are implicitly built into the language. In general, the dependent function type $\prod_{a:A} B$ is written as `(a : A) → B`. In particular, we write the type `id` : $\prod_{A:\mathcal{U}} A \to A$ as `(A : Set) → A → A`.
- Agda allows a shorthand for nested dependent function types of the form $\prod_{a:A} \prod_{a':A} B$. We can write them as `(a a' : A) → B`, instead of having to write `(a : A) → (a' : A) → B`.
- Implicit arguments are declared between curly braces, `{ ... }`. Whenever we call a function with implicit arguments, the type checker will infer them from the context if we decide not to overwrite them explicitely.
- Infix operators can be defined using underscores _.

We have seen that Agda directly provides function types, but it also provides two mechanisms to define new types: ***data declarations*** and ***record types***.

- **Data declarations** can be thought as inductive types. They build the free type (the initial algebra) over a set of constructors; and they are better suited for defining positive types such as the void type, coproducts or natural numbers.
- **Record types** can be thought as generalized dependent n-tuples in which every element of the tuple is labeled and depends on the previous ones. They are better suited for defining negative types such as the unit type or product types. A constructor can be provided for them.

The complete type constructors of MLTT and the majority of types that we will use in this text can be defined in terms of these.

```
- An inductive datatype is determined by a (possibly empty) list of
- constructors.
```

```
data ⊥ : Set where

data _+_ (S : Set) (T : Set) : Set where
 inl : S → S + T
 inr : T → S + T

data ℕ : Set where
 zero : ℕ
 succ : ℕ → ℕ

- A record is determined by a (possibly empty) list of fields that
- have to be filled in order to define an instance of the type.
record ⊤ : Set where
 constructor unit

record _×_ (S : Set) (T : Set) : Set where
 constructor _,_
 field
  fst : S
  snd : T

record  (A : Set)(B : A → Set) : Set where
  constructor _,_
  field
   fst : A
   snd : B fst

- Propositional equality is another example of inductive type,
- namely, the smallest type containing reflexivity.  We define it at
- all levels of the universe hierarchy.
data _==_ {l} {A : Set l} (x : A) : A → Set l where
 refl : x == x
```

Once we have defined these types, we can start profiting from the propositions as types interpretation to write and check mathematical proofs in Agda.

Functions can be defined by a (possibly empty) list of declarations that exhaustively pattern match on all the possible constructors of the type.

```
- Some simple proofs about types in Agda using pattern matching.
- Commutativity of the product.
comm-∗ : {A : Set}{B : Set} → A × B → B × A
comm-∗ (a , b) = (b , a)
```

```
- Associativity of the coproduct.
assocLR-+ : {A B C : Set} → (A + B) + C → A + (B + C)
assocLR-+ (inl (inl a)) = inl a
assocLR-+ (inl (inr b)) = inr (inl b)
assocLR-+ (inr c) = inr (inr c)

- Principle of explosion.  Ex falso quodlibet.
exfalso : {X : Set} → ⊥ → X
exfalso ()

- Projections of the dependent pair type.
```

In particular, we can check our previous proof of the Theorem of Choice defining the two projections from a dependent pair type.

```
- Projections of the dependent pair type.
proj1 : {A : Set} → {B : A → Set} →  A B → A
proj1 (fst , snd) = fst

proj2 : {A : Set} → {B : A → Set} → (p :  A B) → B (proj1 p)
proj2 (fst , snd) = snd

- Theorem of choice
tc : {A B : Set} → {R : A → B → Set} →
 ((x : A) →  B (λ y → R x y)) → ( (A → B) (λ f → (x : A) → R x (f x)))
tc {A} {B} {R} g = (λ x → proj1 (g x)) , (λ x → proj2 (g x))
```

The path induction principle follows from the definition of equality as an inductive type and can be used to prove indiscernability of identicals.

```
- Indiscernability of identicals proved by pattern matching.  We
- only have to prove the case in which the equality is reflexivity.
indiscernability : {A : Set} → {C : A → Set} →
 (x y : A) → (p : x == y) → C x → C y
indiscernability _ _ refl = id
```

23.2 type in type

```
- This proof is based on Thorsten Altenkirch notes on
- Computer Aided Formal Reasoning (G53CFR, G54CFR),
- Nottingham University.
```

```
{-# OPTIONS –type-in-type #-}

module Russell where

  - Imports our previous declarations.
  open import Ctlc
  open import Data.Bool

  - Encoding of set-theoretical sets.  This is a definition of
  - a Set indexed by an index type I. This definition crucially
  - uses Set :  Set, as it is a Set taking a Set as an argument.
  - As Set is a reserved word, we use Aro = Set in esperanto.
  data Aro : Set where
   aro : (Index : Set) → (Index → Aro) → Aro

  - Definition of membership.
  - An element is a member of the set if there exists an
  - index pointing to it on the set.
  _∈_ : Aro → Aro → Set
  x ∈ (aro Index a) =  Index (λ i → x == a i)

  _∉_ : Aro → Aro → Set
  a ∉ b = (a ∈ b) → ⊥

  - Paradoxical set R. It contains all the sets that do not
  - contain themselves.  The index type is the type of the
  - set that do not contain themselves.
  R : Aro
  R = aro ( Aro (λ a → a ∉ a)) proj1


  - Lemma 1.  Every set in R does not contain itself.
  lemma1 : {X : Aro} → X ∈ R → X ∉ X
  lemma1 ((X , proofX∉X) , refl) = proofX∉X

  - Lemma 2.  Every set which does not contain itself is in R.
  lemma2 : {X : Aro} → X ∉ X → X ∈ R
  lemma2 {X} proofX∉X = ((X , proofX∉X) , refl)

  - Lemma 3.  R does not contain itself
  lemma3 : R ∉ R
  lemma3 proofR∈R = lemma1 proofR∈R proofR∈R
```

```
- Russell’s paradox.  We have arrived to a contradiction.
russellsparadox : ⊥
russellsparadox = lemma3 (lemma2 lemma3)
```

24

HOMOTOPY TYPE THEORY I: UNIVALENCE

The basic idea of **Homotopy Type Theory** is to think of types as ∞-groupoids. Between two elements of a type $a, b : A$ we can define their identity type $a =_A b$; between two proofs of equality $p, q : a =_A b$ we can define again a new identity type $p =_{a=b} q$; between any two proofs of equality between equalities, we could define again a new identity type, and so on. All of this ∞-groupoid structure arises from the path induction principle.

Homotopy Type Theory (HoTT) adds the univalence axiom and higher inductive types to intensional type theory (ITT).

This presentation of Homotopy Type Theory will follow [Uni13].

24.1 TYPES AS GROUPOIDS

Types and identity types between their elements are groupoids.

```
- Inverse
inv : {A : Set} → {x y : A} → x == y → y == x
inv refl = refl

- Transitivity
trans : {A : Set} → {x y z : A} → x == y → y == z → x == z
trans refl refl = refl

- Neutrality of refl
neutr-refl-L : {A : Set} → {x y : A} → (p : x == y) → p == trans p refl
neutr-refl-R : {A : Set} → {x y : A} → (p : y == x) → p == trans refl p
neutr-refl-L refl = refl
neutr-refl-R refl = refl

- Concatenation of inverses
inv-trans-L : {A : Set} → {x y : A} → (p : x == y) → trans (inv p) p == refl
```

```
inv-trans-R : {A : Set} → {x y : A} → (p : x == y) → trans p (inv p) == refl
inv-trans-L refl = refl
inv-trans-R refl = refl

- The inverse is an involution
inv-involution : {A : Set} → {x y : A} → (p : x == y) → inv (inv p) == p
inv-involution refl = refl
```

Functions can be seen as functors between groupoids.

```
- Application of functions to paths
ap : {A B : Set} → {x y : A} → (f : A → B) → x == y → f x == f y
ap f refl = refl

- Functions act over equalities as applicative functors
ap-funct-trans : {A B C : Set} → {x y z : A} →
 (f : A → B) → (g : A → B) → (p : x == y) → (q : y == z) →
 ap f (trans p q) == trans (ap f p) (ap f q)
ap-funct-trans f g refl refl = refl

ap-funct-inv : {A B : Set} → {x y : A} → (f : A → B) → (p : x == y) →
 ap f (inv p) == inv (ap f p)
ap-funct-inv f refl = refl

ap-funct-comp : {A B C : Set} → {x y : A} → (g : B → C) → (f : A → B) → (p : x == y) →
 ap g (ap f p) == ap (g ∘ f) p
ap-funct-comp g f refl = refl

ap-funct-id : {A B : Set} → {x y : A} → (p : x == y) → ap id p == p
ap-funct-id refl = refl
```

> Finding a type-theoretic description of this behavior (that is, introduction, elimination and computation rules which comport with Gentzen's Inversion Principle) is an open problem.

Lemma 8 (Transport)**.** *Given a parametrized type family $P : A \to \mathcal{U}$, for every $p : x =_A y$, there exists a function $p_* : P(x) \to P(y)$.*

24.2 homotopies and quasi-inverses

Definition 118 (Homotopy)**.** A **homotopy** between two sections of a fibration $f, g\colon \prod_{x:A} P(x)$ is defined as

$$(f \sim g) :\equiv \prod_{x:A}(f(x) = g(x)).$$

Definition 119 (Quasi-inverse)**.** A **quasi-inverse** of a function $f : A \to B$ is a function $g : B \to A$ and two homotopies $\alpha : f \circ g \sim \mathrm{id}$ and $\beta : g \circ f \sim \mathrm{id}$. More formally, the type of quasi-inverses of f can be described as

$$\sum_{g:B\to A} (f \circ g \sim \mathrm{id}) \times (g \circ f \sim \mathrm{id})$$

- Equivalences
  - Type of equivalences
  - Equivalence of types The type of *equivalences between two types* is defined as
    $$(A \simeq B) :\equiv \sum_{f:A\to B} \mathrm{isequiv}(f)$$
  - Equivalence is an equivalence relation

24.3 higher-groupoid structure of types

The higher-groupoid structure of types manifests itself also in type constructors and how them relate to equality types. In general, we will be able to prove that equalities between type constructors can be characterized by equalities between the arguments of their constructors; but there are two exceptions

- the axiom of **function extensionality** will be needed to prove the desired theorem for Π-Types, and
- the **univalence** axiom will be needed to prove the desired theorem for $\mathcal{U}$-Types.

- Function extensionality
  Definition 120 (Function extensionality)**. Function extensionality** states that the function
  $$(f = g) \to \prod_{x:A} \Big(f(x) =_{B(x)} g(x) \Big)$$
  is an equivalence. In particular, the following function is its quasi-inverse
  $$\mathrm{funext} : \left(\prod_{x:A} f(x) = g(x) \right) \to (f = g).$$

24.4 univalence axiom

We know that, for any pair of types $A, B : \mathcal{U}$, there exists a function of type idtoeqv : $(A =_{\mathcal{U}} B) \to (A \sim B)$, defined

Definition 121 (Univalence)**.** For any pair of types $A, B : \mathcal{U}$, ideqv : $(A =_{\mathcal{U}} B) \to (A \sim B)$ is an equivalence. In particular, $A = B$ is equivalent to $A \sim B$.

- Univalent universes

24.5 n-types

A *type-theoretical set* will be a type with no associated higher groupoid structure. That is, it is a discrete groupoid, and any two parallel paths are equal.

Definition 122 (Sets)**.** A type A is a **set** (or **0-type**) when the following type is inhabited

$$\texttt{isSet}(A) :\equiv \prod_{x,y:A} \prod_{p,q:x=y} (p = q).$$

We can generalize this definition to higher dimensions

- a **0-type** is a type with no non-trivial paths
- a **1-type** is a type with no non-trivial paths between paths, every two paths $r, s : p = q$ between types $p, q : x = y$ must be equal, $r = s$;
- a **2-type** is a type with no non-trivial paths between paths between paths;
- ... and so on.

In particular, it can be proved that every **n-type** is an **(n+1)-type**.

24.6 higher inductive types

25

HOMOTOPY TYPE THEORY II: MATHEMATICS

Part VI

CONCLUSIONS

Part VII

APPENDICES

BIBLIOGRAPHY

[AB17] Steven Awodey and Andrej Bauer. Lecture notes: Introduction to categorical logic, 2017.

[Bar84] H.P. Barendregt. *The lambda calculus: its syntax and semantics*. Studies in logic and the foundations of mathematics. North-Holland, 1984.

[Bar92] H. P. Barendregt. Handbook of logic in computer science (vol. 2). chapter Lambda Calculi with Types, pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992.

[Bar94] Henk Barendregt Erik Barendsen. Introduction to lambda calculus, 1994.

[CAB$^+$86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[CCHM16] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. *CoRR*, abs/1611.02108, 2016.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic, Volume I.* North-Holland, 1958. Second printing 1968.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[Cro75] J. N. Crossley. *Reminiscences of logicians*, pages 1–62. Springer Berlin Heidelberg, Berlin, Heidelberg, 1975.

[Cur34] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):584–590, 1934.

[Cur46] Haskell B. Curry. The paradox of kleene and rosser. *Journal of Symbolic Logic*, 11(4):136–137, 1946.

[dB72] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381 – 392, 1972.

[EM42] Samuel Eilenberg and Saunders MacLane. Group extensions and homology. *Annals of Mathematics*, 43(4):757–831, 1942.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1989.

[HHJW07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: Being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, page 1. Microsoft Research, April 2007.

[HM96] Graham Hutton and Erik Meijer. Monadic parser combinators, 1996.

[HM98] Graham Hutton and Erik Meijer. Monadic parsing in haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[HS08] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators: An Introduction*. Cambridge University Press, New York, NY, USA, 2 edition, 2008.

[Hug89] J. Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, April 1989.

[Jup] Jupyter Development Team. Jupyter notebooks. a publishing format for reproducible computational workflows.

[Kam01] Fairouz Kamareddine. Reviewing the classical and the de bruijn notation for lambda-calculus and pure type systems. *Logic and Computation*, 11:11–3, 2001.

[Kas00] Ryo Kashima. A proof of the standardization theorem in lambda-calculus. page 6, 09 2000.

[KR35] S. C. Kleene and J. B. Rosser. The inconsistency of certain formal logics. *Annals of Mathematics*, 36(3):630–636, 1935.

[Kun11] K. Kunen. *Set Theory*. Studies in logic. College Publications, 2011.

[Lan78] Saunders Mac Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer New York, 1978.

[Law64] F. William Lawvere. An Elementary Theory of the Category of Sets. *Proceedings of the National Academy of Sciences of the United States of America*, 52(6):1506–1511, December 1964.

[Lei01] Daan Leijen. Parsec, a fast combinator parser. Technical Report 35, Department of Computer Science, University of Utrecht (RUU), October 2001.

[LS09] F. William Lawvere and Stephen H. Schanuel. *Conceptual Mathematics: A First Introduction to Categories*. Cambridge University Press, New York, NY, USA, 2nd edition, 2009.

[04] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[McB17] Conor McBride. Cs410 advanced functional programming. https://github.com/pigworker/CS410-17, 2017.

[Mcc91] William W. Mccune. Single axioms for groups and abelian groups with various operations. In *Preprint MCS-P270-1091, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL*, 1991.

[MM94] I. Moerdijk and S. MacLane. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Universitext. Springer New York, 1994.

[O'S16] Bryan O'Sullivan. The attoparsec package. http://hackage.haskell.org/package/attoparsec, 2007–2016.

[P$^+$03] Simon Peyton-Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. http://www.haskell.org/definition/.

[Pol95] Robert Pollack. Polishing up the tait-martin-löf proof of the church-rosser theorem, 1995.

[Sel13] Peter Selinger. Lecture notes on the lambda calculus. Expository course notes, 120 pages. Available from 0804.3434, 2013.

[Tai67] W. W. Tait. Intensional interpretations of functionals of finite type i. *The Journal of Symbolic Logic*, 32(2):198–212, 1967.

[Tur37] A. M. Turing. Computability and λ-definability. *The Journal of Symbolic Logic*, 2(4):153–163, 1937.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[Wad85] Philip Wadler. How to replace failure by a list of successes. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pages 113–128, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[Wad90] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 61–78, New York, NY, USA, 1990. ACM.

[Wad15] Philip Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, November 2015.