CATEGORY THEORY AND LAMBDA CALCULUS

MARIO ROMÁN

UNIVERSIDAD
DE GRANADA

Trabajo Fin de Grado

Doble Grado en Ingeniería Informática y Matemáticas

Tutores
Jesús García Miranda
Pedro A. García-Sánchez

FACULTAD DE CIENCIAS
E.T.S. INGENIERÍAS INFORMÁTICA Y DE TELECOMUNICACIÓN

Granada, a junio de 2018

CONTENTS

ABSTRACT

This is the abstract.

Part I

LAMBDA CALCULUS

1

UNTYPED λ-CALCULUS

How should we define functions? 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 equal if they map equal inputs to equal outputs; and a function its completely determined by what its outputs are. This vision is called ***extensional***.

From a computational point of view, this perspective could be incomplete in some cases. Computationally, we usually care not only about the result but, crucially, about *how* can it be computed. Classically in computer science, *functions are formulae*; and two functions mapping equal inputs to equal outputs need not to be equal. E.g., 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 discovered 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 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 which can be directly applied to any expression. It even allows functions to be applied to themselves, a notion which would be troublesome[1] in our usual set-theoretical foundations. In exchange, it presents other problems such as non-terminating functions.

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

1 : 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.

1.1 THE UNTYPED λ-CALCULUS

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

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, $\lambda x.M$ would be a function taking x as an argument and returning M, which would be a term where x could appear in.

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)**.** The **λ-terms** are defined inductively knowing that

- 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.

Equivalently, they can be also 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, i.e., MNP will always mean $(MN)P$. We also take the λ-abstraction as having the lowest precedence, e.g., $\lambda x.MN$ 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 indepen-

dently. For example, in $(\lambda x.x)(\lambda x.x)$, which evaluates to $(\lambda x.x)$, x appears two times as a variable with two different meanings.

Any ocurrence of a variable x inside the *scope* of a lambda is said to be ***bound***; and any not bound variable 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 $x^2 + x$ in the place of 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)

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

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

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

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

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

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 properties of the term. This idea appears recurrently on mathematics; e.g., 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 which is an equivalence relation, i.e.,

- 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 $\to_\beta$, 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

$$(\lambda x.M)N \to_\beta M[N/x],$$

and some congruence rules that preserve the structure of λ-terms, such as

- $M \to_\beta M'$ implies $MN \to_\beta M'N$ and $NM \to_\beta NM'$;
- $M \to_\beta M'$ implies $\lambda x.M \to_\beta \lambda x.M'$.

The reflexive transitive closure of $\to_\beta$ 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

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

We define single-step $\beta\eta$-reduction as the union of β-reduction and η-reduction. This will be written as $\to_{\beta\eta}$, 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

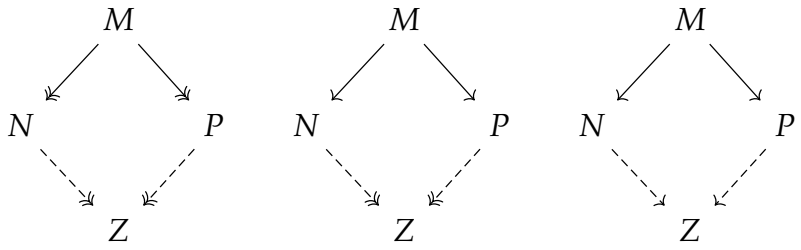
As we said earlier, it is not possible in general to evaluate a λ-term into a canonical, non-reducible term. However, we will be able to prove that, in the cases where it exists, it will be unique. This property is a consequence of a sightly more general one, ***confluence***, which can be defined in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

Definition 7 (Confluence)**.** A relation $\to$ is **confluent** if, given its reflexive transitive closure $\twoheadrightarrow$, $M \twoheadrightarrow N$ and $M \twoheadrightarrow P$ imply the existence of some Z 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



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, as a naive attempt of proof by induction would attempt.

The statement of $\twoheadrightarrow_\beta$ and $\twoheadrightarrow_{\beta\eta}$ being confluent is what we are going to 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, $\rhd$ as the smallest relation satisfying that, assuming $P \rhd P'$ and $N \rhd N'$, the following properties of

- 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

- $x^* = x$;
- $(PN)^* = P^*N^*$;
- $((\lambda x.P)N)^* = P^*[N^*/x]$;
- $(\lambda x.N)^* = \lambda x.N^*$;
- $(\lambda x.Px)^* = P^*$, 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 $\to_{\beta\eta}$. □

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 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 term is weakly normalizing if and only if it has a normal form. Strong normalization implies also 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

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

is weakly normalizing but not strongly normalizing. Its normal form is

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

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, 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 $\to_l$; $\twoheadrightarrow_l$ 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 theorem, the existence of an optimal beta reduction strategy, in the sense that it will always find the normal form if it 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 relation given by the head reduction of the application and it is defined by

- $A \twoheadrightarrow_h A$, reflexivity.
- $(\lambda x.A_0)A_1 \ldots \twoheadrightarrow_h A_0[A_1/x] \ldots$, head β-reduction.
- $A \twoheadrightarrow_h B \twoheadrightarrow_h C$ implies $A \twoheadrightarrow_h C$, transitivity.

The second one is the standard reduction and it is defined by

- $M \twoheadrightarrow_h x$ implies $M \twoheadrightarrow_s x$, where x is a variable.
- if $M \twoheadrightarrow_h AB$, $A \twoheadrightarrow_s C$ and $B \twoheadrightarrow_s D$, then $M \twoheadrightarrow_s CD$.
- if $M \twoheadrightarrow_h \lambda x.A$ and $A \twoheadrightarrow_s B$, then $M \to_s \lambda x.B$.

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 $M[P/z] \twoheadrightarrow_s N[Q/z]$.

Now we can prove that $K \twoheadrightarrow_s (\lambda x.M)N$ implies $K \twoheadrightarrow_s M[N/x]$. From the fact that $K \twoheadrightarrow_s (\lambda x.M)$, we know that there must exist P and Q such that $K \twoheadrightarrow_h \lambda PQ$, $P \twoheadrightarrow_s \lambda x.M$ 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 $M \twoheadrightarrow_\beta N$, 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

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

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 "Programming in the untyped λ-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 the 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 the λ-calculus and logic arises. This connection will be discussed in the next chapter.

In programming languages, types signal 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 the λ-calculus 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.

The **simply typed λ-calculus** (STLC) 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. If 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 the 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 of 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 THE SIMPLY TYPED λ-CALCULUS

We will now define the terms of the 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 BNF

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

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 typing assumptions $x_1 : A_1, \ldots, x_n : A_n$, 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. The (var) rule simply makes explicit the type of a variable from the context. That is, a context that assumes $x : A$ can be written as $\Gamma, x : A$; and we can trivially deduce from it that $x : A$.

1 : In particular, we can not apply a term of type $A \to B$ to a term of type $A \to B$; a term of type A was expected.

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

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.

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

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.

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

Definition 20. A term is **typeable** if a typing judgment for the term is derivable.

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

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

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, at 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 which 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 STLC with two basic types.

The term's type is defined as an intrinsic property of the term; and the same term has to always be 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 allow us to 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 allow 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 (Type variables)**.** Given a infinite numerable set of *type variables*, we define **parametric types** or *type-schemes* inductively as

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

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. It can be applied to a type template as $\overline{\psi}$ by recursion and knowing that

- $\overline{\psi}\iota = \iota$,
- $\overline{\psi}\sigma = \psi\sigma$,
- $\overline{\psi}(A \to B) = \overline{\psi}A \to \overline{\psi}B$.

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

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 instance, $A \to B$ is more general than its instance $(C \to D) \to B$, where A has been instantiated to $C \to D$. An interesting property of STLC is that every type has a most general type, called its *principal type*. This is 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; i.e., 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. *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. $\texttt{unify}(x; x) = \mathrm{id}$ *and* $\texttt{unify}(\iota, \iota) = \mathrm{id}$*;*
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 of types have no unifier if and only if $\texttt{unify}(A, B)$ *fails.*

Proof. It is easy to notice that, by structural induction, 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* $\texttt{typeinfer}(M, B)$*, defined as follows, finds the most general substitution* σ *such that* $x_1 : \sigma A_1, \ldots, x_n : \sigma A_n \vdash M : \overline{\sigma} B$ *is a valid typing judgment if it exists; and fails otherwise.*

1. $\texttt{typeinfer}(x_i : A_i, \Gamma \vdash x_i : B) = \texttt{unify}(A_i, B)$;
2. $\texttt{typeinfer}(\Gamma \vdash MN : B) = \overline{\varphi} \circ \psi$*, where* $\psi = \texttt{typeinfer}(\Gamma \vdash M : x \to B)$ *and* $\varphi = \texttt{typeinfer}(\overline{\psi}\Gamma \vdash N : \overline{\psi} x)$ *for a fresh type variable* x*.*
3. $\texttt{typeinfer}(\Gamma \vdash \lambda x.M : B) = \overline{\varphi} \circ \psi$ *where* $\psi = \texttt{unify}(B; z \to z')$ *and* $\varphi = \texttt{typeinfer}(\overline{\psi}\Gamma, x : \overline{\psi} z \vdash M : \overline{\psi} z')$ *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 $\texttt{typeinfer}(x_1 : A_1, \ldots, x_n : A_n \vdash M : B)$, where $x_1, \ldots, x_n$ are the free variables on M and $A_1, \ldots, A_n, B$ 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 $x_1 : U_1, \ldots, x_n : U_n$, and reducible terms $u_1 \in \mathtt{RED}_{U_1}, \ldots, u_n \in \mathtt{RED}_{U_2}$, we know that*

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

Proof. We call $\tilde{t} = t[u_1/x_1][u_2/x_2]\ldots[u_n/x_n]$ 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 the STLC, 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 the STLC and every computation ends. We know, however, that the Halting Problem is unsolvable, so the STLC 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

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

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.

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

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

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

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

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

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]. It relationship with the STLC will be made explicit on the next section.

We will use the logical binary connectives $\to, \wedge, \vee$, and two given propositions, $\top, \bot$ representing truth and falsity. 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*.

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

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.

$$\dfrac{\dfrac{\dfrac{[A]}{\vdots}\,1}{B}\qquad \overset{2}{\vdots}}{\dfrac{A \to B \qquad A}{B}} \quad\Longrightarrow\quad \dfrac{\overset{2}{\vdots}}{\dfrac{A}{\dfrac{\overset{1}{\vdots}}{B}}}$$

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

$$\dfrac{\dfrac{\overset{1}{\vdots} \qquad \overset{2}{\vdots}}{\dfrac{A \qquad B}{A \wedge B}}}{A} \quad\Longrightarrow\quad \dfrac{\overset{1}{\vdots}}{A}$$

and, similarly, on the other side as

$$\dfrac{\dfrac{\dfrac{\vdots^{1}}{A}\quad\dfrac{\vdots^{2}}{B}}{A \wedge B}}{B}\vdots \quad\Longrightarrow\quad \dfrac{\vdots^{2}}{B}\vdots$$

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

$$\dfrac{\dfrac{\vdots^{1}}{A}}{A+B}\quad\dfrac{[A]^{2}}{\vdots}{C}\quad\dfrac{[B]^{3}}{\vdots}{C}\;\Big/\;C\vdots \quad\Longrightarrow\quad \vdots^{1}\,A\;\vdots^{2}\,C\vdots$$

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

$$\dfrac{\dfrac{\vdots^{1}}{B}}{A+B}\quad\dfrac{[A]^{2}}{\vdots}{C}\quad\dfrac{[B]^{3}}{\vdots}{C}\;\Big/\;C\vdots \quad\Longrightarrow\quad \vdots^{1}\,B\;\vdots^{3}\,C\vdots$$

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 STLC and natural deduction, the correspondence starts when we describe the following isomorphism 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 ($\bot$)

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.

Under this interpretation, ***simplification rules are precisely the 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 2 (Curry-Howard 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*

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

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

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

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

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

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 on intuitionistic logic. This example could be checked and simplified by this interpreter as it is shown in image 1.

```
1  :types on
2
3  # Evaluates this term
4  \a.((\c.Case c Of inr; inl)(INL a))
5
6  # Draws the deduction tree
7  @ \a.((\c.Case c Of inr; inl)(INL a))
8
9  # Simplifies the deduction tree
10 @@ \a.((\c.Case c Of inr; inl)(INL a))
```

evaluate

```
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

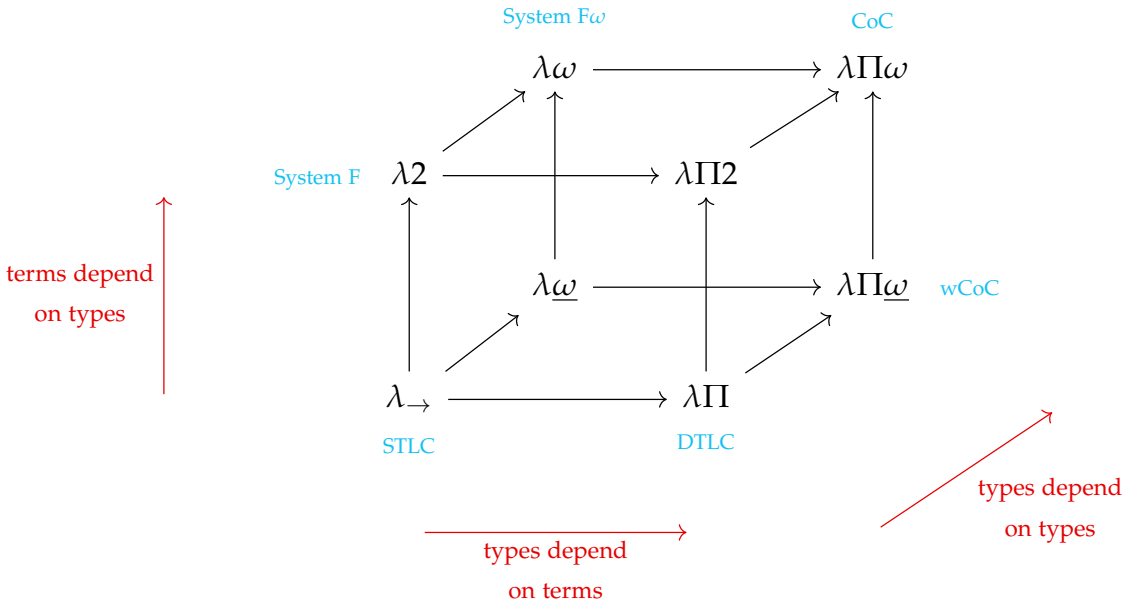
1. **parametric polymorphism**, terms that depend on types. This is achieved via universal quantification over types. It allows type variables and binders for them. An example is the following parametric identity function

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

   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** ($\lambda_{\to}$);
- **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

PROGRAMMING ENVIRONMENT

5.1 THE HASKELL PROGRAMMING LANGUAGE

Haskell is the purely functional programming language of our choice to implement Mikrokosmos. Its 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 on [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

- **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.
- **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], Hughes argued for the benefits of a lazy functional language, which could solve the efficiency problem.
- **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 any other side-effects by relying on the evaluation order. This helps modularity of the code, testing and verfication.

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

- **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 derive directly from λ-calculus.
- based on **System F**ω 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 the ability to activate extensions that implement dependent types.

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

```haskell
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 to define common interfaces for different types. In the following example, we define a `Monad` as a type with a suitably typed `return` and `bind` operators.

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

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

```haskell
instance Monad [] where
  return x = [x]
  xs >>= f = concat (map f xs)
```

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.

```haskell
pythagorean = do
  a <- [1..]
  b <- [1..a]
  c <- [1..b]
  guard (a^2 == b^2 + c^2)
  return (a,b,c)
```

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.

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

5.2 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 in for Haskell software.

Dependencies and packaging details for Mikrokosmos are specified in the file `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.

5.3 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 testing 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.

```
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 deBruijn expression keeps is meaning when is translated into a λ-term.

```
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 correct interpretation of a 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 b))))) ⇒ 6 :: (A → A) → A → A
```

5.4 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.

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.

6

IMPLEMENTATION OF λ-EXPRESSIONS

6.1 DE BRUIJN INDEXES

Nicolaas Govert **De Bruijn** proposed in [dB72] a way of defining λ-terms modulo α-conversion based on indices. The main idea 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 but preserve 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

$$\texttt{Exp} ::= \mathbb{N} \mid (\lambda\ \texttt{Exp}) \mid (\texttt{Exp}\ \texttt{Exp})$$

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.

6.2 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.

```
-- | 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 then defined using this `subs` function.

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

6.3 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
-- into 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 of applications is trivial.
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.

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

```haskell
-- | 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)
```

6.4 EVALUATION

As we proved on Corollary 1, the leftmost reduction strategy will find the leftmost reduction strategy if it exists. Consequently, we will implement it using a function that simply applies the leftmost possible reductions at each step. 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.
- 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 $(\lambda x.xx)(\lambda x.xx)$.

6.5 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 bottom type;
- product types;
- union types;
- and function types.

```haskell
-- | 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.

```haskell
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 all the computation. The algorithm is exactly the same that was defined in Lemma 4.

```haskell
-- | 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.

```haskell
-- | 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.

```
-- | Type inference of a lambda expression.
typeinference :: Exp -> Maybe Type
typeinference e = normalize <$>
  (typeinfer variables emptyctx e (Tvar 0) <*> pure (Tvar 0))
```

The complete code can be found on the [BROKEN LINK: *Mikrokosmos complete code].

7

USER INTERACTION

7.1 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 is discussed firstly in [Wad85], and later in [Wad90] and [HM98]. The parser type is defined as a function taking a `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
newtype Parser a = Parser (String -> [(a,String)])

parse :: Parser a -> String -> [(a,String)]
parse (Parser p) = p

instance Monad Parser where
  return x = Parser (\s -> [(x,s)])
  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 as

```haskell
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

```haskell
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.

7.2 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.

7.3 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.

```
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.

```
-- | 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
```

7.4 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.

```
data Ski = S | K | I | Comb Ski Ski
```

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)
```

8

USAGE

8.1 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 on 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

8.2 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.

The Figure 2 is an example session on the mikrokosmos interpreter.

8.3 JUPYTER KERNEL

The **Jupyter Project** [Tea] 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>')
```

Figure 2: Mikrokosmos interpreter session.

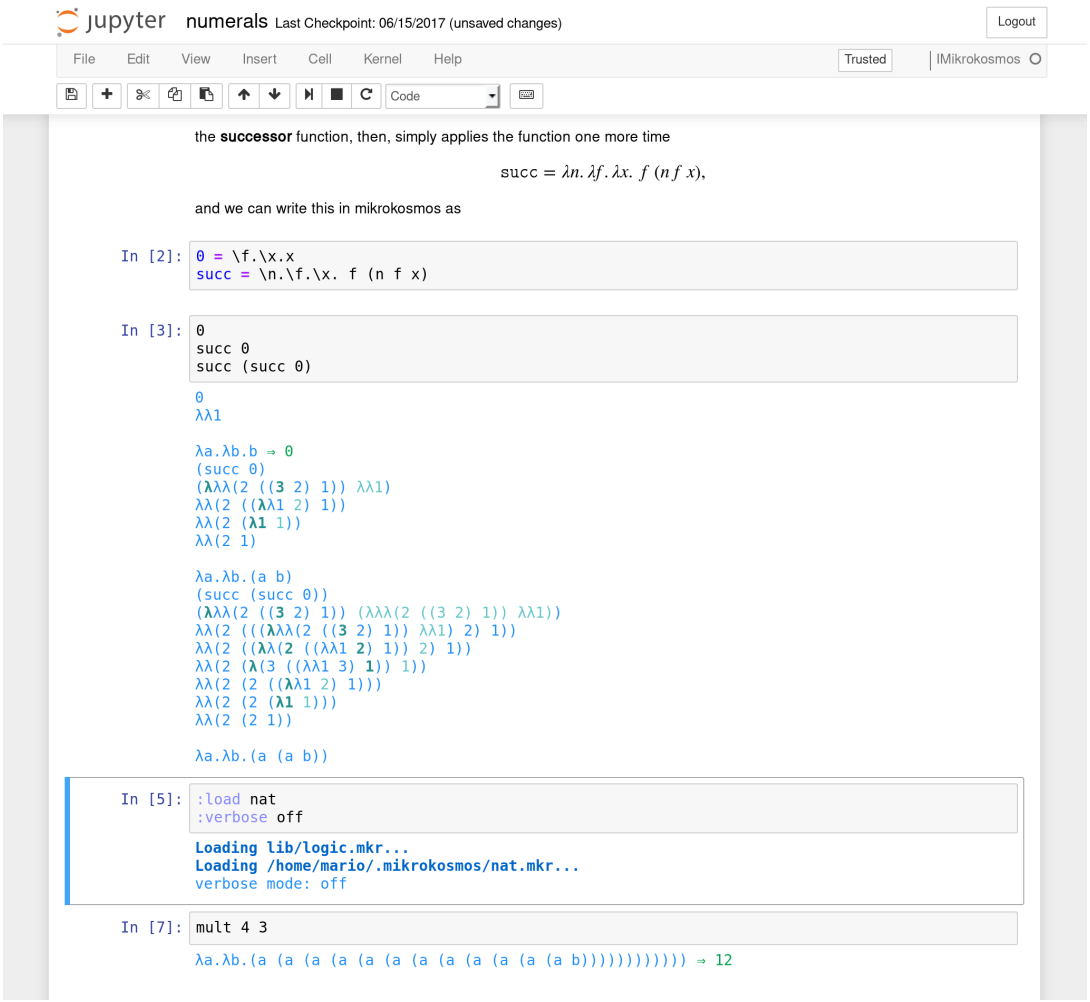


Figure 3: Jupyter notebook Mikrokosmos session.

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

8.4 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"]},
```

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/

```
    // Operators
    {regex: /[=!]+/,
    token: "operator"},
    ],
    meta: {
        dontIndentStates: ["comment"],
        lineComment: "#"
    }
}
```

8.5 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.

8.6 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

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

```
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

```
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.

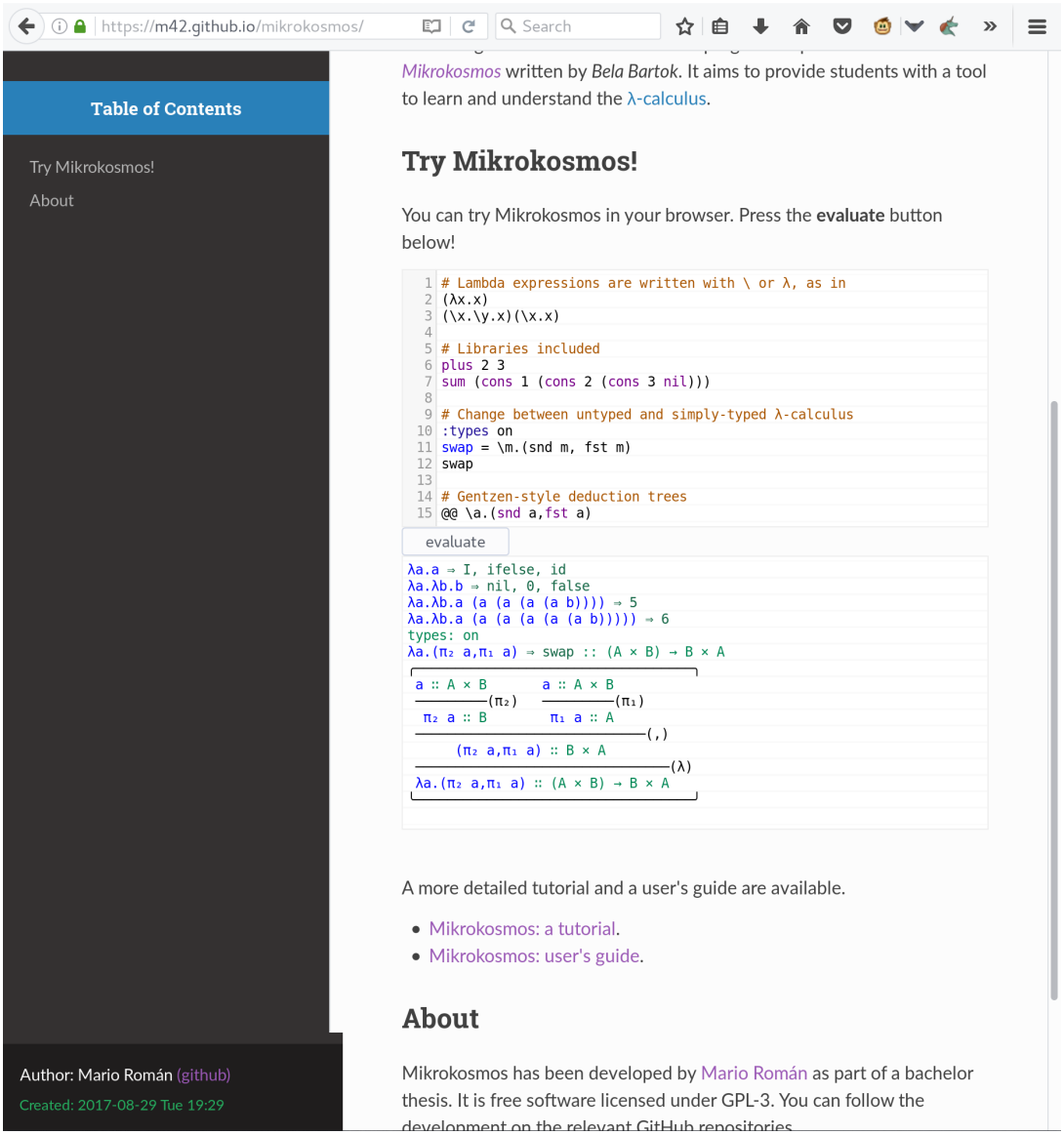


Figure 4: Mikrokosmos embedded into a web page.

9

PROGRAMMING IN THE UNTYPED λ-CALCULUS

This section explains how to use the 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 any 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 denoted with the => symbol, as in

```
compose id id
---> λa.a ⇒ id
```

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

```
alwaysid = 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, the **hom-tensor adjunction**

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

or the definition of exponentials.

9.2 A TECHNIQUE ON INDUCTIVE DATA ENCODING

Over this presentation, we will implicitly use a technique on the majority of our data encodings which allows us to write an encoding for any algebraically inductive generated data. This technique is used without comment on [Sel13] and is the basic of what is called the **Church encoding**.

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

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

Or, in general

$$\texttt{T} ::= C_1 \mid C_2 \mid C_3 \mid \dots$$

We do not have any possibility of encoding constructors on λ-calculus. Even if we had, they would have, in theory, no computational content; the application of constructors 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. The λ-calculus would need to be extended with additional syntax for every new type.

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

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

In general, any instance of the type `T` 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 compute by instantiating this algebra 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, i.e.,

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

We can test this interpretation on the interpreter to get

```
true id const
--- => id

false id const
--- => const
```

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

```
ifelse = \b.b
(ifelse true) id const
--- => id
(ifelse false) id const
--- => false
```

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
--- => false
```

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
--- => false
```

```
4 not true
--- => true

double = \n.\s.\z.n (compose s s) z
double 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
--- => 3
mult 2 4
--- => 8
```

9.5 THE PREDECESSOR FUNCTION AND PREDICATES ON NUMBERS

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
--- => true
iszero 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

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.

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

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)))
--- => 6
all (cons true (cons true (cons true nil)))
--- => 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)
--- => 9
length (filter (leq 2) mylist)
--- => 2
```

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 on intuitionistic 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 every term before its evaluation. The type will then be displayed after the result of the computation.

Example 3 (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
---> λa.a ⇒ id, I, ifelse :: A → A

true
false
---> λa.λb.a ⇒ K, true :: A → B → A
---> λa.λb.b ⇒ nil, 0, false :: A → B → B

0
1
2
---> λa.λb.b ⇒ nil, 0, false :: A → B → B
```

```
---> λa.λb.(a b) ⇒ 1 :: (A → B) → A → B
---> λa.λb.(a (a b)) ⇒ 2 :: (A → A) → A → A

S
K
---> λa.λb.λc.((a c) (b c)) ⇒ S :: (A → B → C) → (A → B) → A → C
---> 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 which 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 4 (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 this point, we have only used the function type. We are working on the implicational fragment of the STLC we described on the first typing rules. We are now going to extend the type system in the same sense we extended the STLC. The following types are added to the type system

Type	Name	Description
$\rightarrow$	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 addition 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 5 (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
---> λa.((SND a),(FST a)) ⇒ swap :: (A × B) → B × A

caseof (inl 1) pred (mult 2)
caseof (inr 1) pred (mult 2)
---> λa.λb.b ⇒ nil, 0, false :: A → B → B
---> λa.λb.(a (a b)) ⇒ 2 :: (A → A) → A → A

\x.((abort x),(absurd x))
---> 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
---> (INR UNIT) ⇒ bfalse :: A + ⊤
bnot bfalse
---> (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 ON INTUITIONISTIC LOGIC

Under the logical interpretation of Mikrokosmos, we can transcribe proofs in intuitionistic logic to λ-terms and check them on the interpreter.

Theorem 8. *In intuitionistic logic, the double negation of the LEM 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
---> λa.(ABSURD (a (INR λb.(a (INL b))))) :: ((A + (A → ⊥)) → ⊥) → ⊥
```

whose type is precisely $((\mathtt{A} + (\mathtt{A} \to \bot)) \to \bot) \to \bot$.

Part III

CATEGORY THEORY

11

CATEGORIES

11.1 DEFINITION OF CATEGORY

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 some associativity and identity laws will apply.

Definition 29 (Category)**.** A **category** $\mathcal{C}$, as defined in [Lan78], is given by

- $\mathcal{C}_0$, a collection[1] whose elements are called **objects**[2], and
- $\mathcal{C}_1$, a collection whose elements are called **morphisms**.

Every morphism $f \in \mathcal{C}_1$ has two objects assigned: 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 $g \circ f \colon A \to C$. Morphism composition is a binary associative operation with identity elements $\mathrm{id}_A \colon A \to A$, that is

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

Definition 30 (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 the hom-set $\mathrm{end}(A) = \hom(A, A)$.

Sometimes, when considering a hom-set, it will be useful to explicitly specify the category on which we are working as $\hom_{\mathcal{C}}(A, B)$.

1 : We use the term *collection* to denote some unspecified formal notion of compilation of "things" 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.

2 : We will sometimes write this class of objects of a category $\mathcal{C}$ as $\mathrm{obj}(\mathcal{C})$, but it is common to simply use $\mathcal{C}$ to denote it.

Definition 31 (Small and locally small categories)**.** A category is said to be **small** if the collections $\mathcal{C}_0, \mathcal{C}_1$ of objects and morphisms are both sets (instead of proper classes). It is said to be **locally small** if every hom-set is actually a set.

11.2 MORPHISMS

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

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

exists.

We call **automorphisms** to the endomorphisms which are also 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-side 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.

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 which 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.

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

By virtue of Proposition 2, a morphism which is both a retraction and a section is an isomorphism.

11.3 TERMINAL OBJECTS, PRODUCTS AND COPRODUCTS

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

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

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,

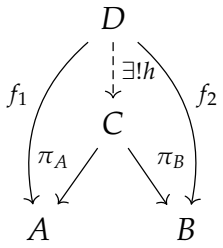
- $f \circ g = \mathrm{id}$,
- $g \circ f = \mathrm{id}$.

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

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,



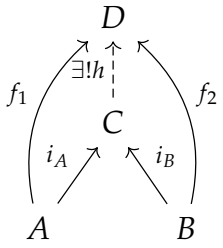
.

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,



.

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 6 (Discrete categories). A category is **discrete** if it has no other morphisms than the identities. A discrete category is uniquely defined by its class of objects and every class of objects defines a category.

Example 7 (Monoids, groups). A single-object category is a **monoid**. [3] A monoid in which every morphism is an isomorphism is a **group**. A **groupoid** is a category (of any number of objects) where all the morphisms are isomorphisms.

Example 8 (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.

Example 9 (The category of sets). The category `Sets` 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.

Example 10 (The category of groups). The category `Grp` is defined as the category with groups as objects and group homomorphisms between them as morphisms.

Example 11 (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 12 (The category of topological spaces). The category `Top` is defined as the category with topologicaal spaces as objects and continuous functions between them as morphisms.

3 : 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.

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*.

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.

12.1 FUNCTORS

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

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

such that

- $F(\text{id}_A) = \text{id}_{FA}$, identities are preserved;
- $F(f \circ g) = Ff \circ Fg$, 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.

Definition 42 (Composition of functors)**.** 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 object and arrow functions of the functors. This composition is trivially associative.

Definition 43 (Identity functor)**.** 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 44 (Full functor)**.** A functor F is **full** if the arrow map is surjective. That is, if every $g : FA \to FB$ is of the form Ff for some morphism $f\colon A \to B$.

Definition 45 (Faithful functor)**.** A functor F is **faithful** if the arrow map 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).

Definition 46 (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.

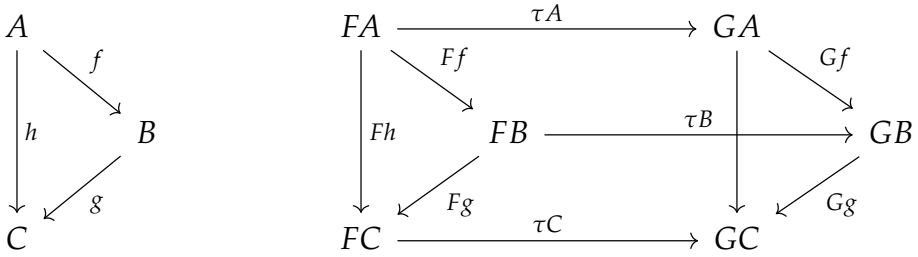
12.2 NATURAL TRANSFORMATIONS

Definition 47 (Natural transformation)**.** A **natural transformation** between two functors with the same domain and codomain, $\alpha\colon F \dot{\to} G$, 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

$$\begin{array}{ccc} C \\ \downarrow f \\ C' \end{array} \qquad \begin{array}{ccc} SC & \xrightarrow{\tau_C} & TC \\ \downarrow Sf & & \downarrow Tf \\ SC' & \xrightarrow{\tau_{C'}} & TC' \end{array}$$

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

It is also said that the family of morphisms is *natural* in its parameter. This naturality property is what allows us to translate a commutative diagram from a functor to another.

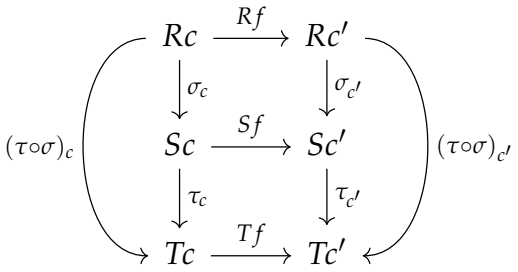


Definition 48 (Natural isomorphism)**.** A **natural isomorphism** is a natural transformation in which every component, every morphism of the parameterized family, is invertible.

The inverses of a natural transformation form another natural transformation, whose naturality follows from the naturality of the original transformation.

12.3 COMPOSITION OF NATURAL TRANSFORMATIONS

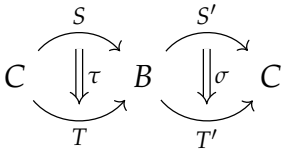
Definition 49 (Vertical composition of natural transformations)**.** The **vertical composition** of two natural transformations $\tau : S \to T$ and $\sigma : R \to S$, denoted by $\tau \cdot \sigma$ is the family of morphisms defined by the objectwise composition of the components of the two natural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



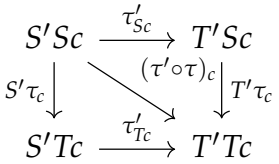
Proposition 5 (Vertical composition is a natural transformation)**.** *The vertical composition of two natural transformations is in fact a natural transformation.*

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



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



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

$$\begin{array}{ccccc} S'Sc & \xrightarrow{S'\tau} & S'Tc & \xrightarrow{\tau'} & T'Tc \\ \downarrow{\scriptstyle S'Sf} & & \downarrow{\scriptstyle S'Tf} & & \downarrow{\scriptstyle T'Tf} \\ S'Sb & \xrightarrow{S'\tau} & S'Tb & \xrightarrow{\tau'} & T'Tb \end{array}$$

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

13

CONSTRUCTIONS ON CATEGORIES

13.1 OPPOSITE CATEGORIES AND CONTRAVARIANT FUNCTORS

Definition 51 (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 a *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 52 (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.

13.2 PRODUCT CATEGORIES

Definition 53 (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 $f \colon c \to c'$ and $g \colon d \to d'$ 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

$$(f', g') \circ (f, g) = (f' \circ f, g' \circ g).$$

We also define **projection functors** $P\colon \mathcal{C} \times \mathcal{D} \to \mathcal{C}$ 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 54 (Product of functors)**.** The **product functor** of two functors $F\colon \mathcal{C} \to \mathcal{C}'$ and $G\colon \mathcal{D} \to \mathcal{D}'$ is a functor $F \times G\colon \mathcal{C} \times \mathcal{D} \to \mathcal{C}' \times \mathcal{D}'$ 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

$$\begin{array}{ccccc} \mathcal{C} & \xleftarrow{P} & \mathcal{C} \times \mathcal{D} & \xrightarrow{Q} & \mathcal{D} \\ \downarrow{\scriptstyle F} & & \vdots{\scriptstyle F \times G} & & \downarrow{\scriptstyle G} \\ \mathcal{C}' & \xleftarrow[P']{} & \mathcal{C}' \times \mathcal{D}' & \xrightarrow[Q']{} & \mathcal{D}' \end{array}$$

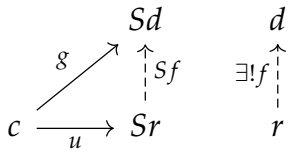
In this sense, the $\times$ operation is itself a functor acting on objects and morphisms of the `Cat` category of all categories.

14

UNIVERSALITY

14.1 UNIVERSAL ARROWS

Definition 55 (Universal arrow)**.** A **universal arrow** from c to S is an arrow $u\colon c \to Sr$ such that for every $c \to Sd$ exists a unique $r \to d$ making this diagram commute



.

Proposition 7 (Universality in terms of hom-sets)**.** *The arrow $u\colon c \to Sr$ is universal iff $f \mapsto Sf \circ u$ 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. Bijection follows from the definition of universal arrow, and naturality follows from $S(gf) \circ u = Sg \circ Sf \circ u$.

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

14.2 REPRESENTABILITY

Definition 56 (Representation of a functor)**.** A **representation** of $K\colon D \to \texttt{Sets}$ is a natural isomorphism

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

A functor is *representable* if it has a representation. An object r is called a *representing object*. D must have small hom-sets.

Proposition 8 (Representations in terms of universal arrows)**.** *If $u\colon * \to Kr$ is a universal arrow for a functor $K\colon D \to \texttt{Sets}$, 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 $D(r, -) \cong \mathtt{Sets}(*, K-)$. But every natural isomorphism of this kind is a [BROKEN LINK: *Universal arrows as natural bijections]. □

14.3 YONEDA LEMMA

Lemma 7 (Yoneda Lemma)**.** *For any $K\colon D \to \mathtt{Sets}$ 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, $\alpha_r 1_r$.

Proof. □

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

Proof. Using Yoneda Lemma, 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 h_* by naturality. □

Proposition 9 (Addendum to the Yoneda Lemma)**.** *The bijection on the Yoneda Lemma is a natural isomorphism between two $\mathtt{Sets}^D \times D \to \mathtt{Sets}$ functors.*

Proof. □

Definition 57. In the conditions of Yoneda Lemma, the **Yoneda functor**, $Y\colon D^{op} \to \mathtt{Sets}^D$, is defined with the arrow function

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

Proposition 10. *The Yoneda functor is full and faithful.*

Proof. □

15

ADJOINTS

15.1 ADJUNCTIONS

Definition 58 (Adjunction)**.** An **adjunction** from categories X to A is a pair of functors $F\colon X \to A$, $G\colon A \to X$ with a natural bijection

$$\varphi\colon \hom(Fx, a) \cong \hom(x, Ga),$$

natural in both $x \in X$ and $a \in A$. We write it as $F \dashv G$.

Definition 59 (Unit and counit of an adjunction)**.** An adjunction determines a **unit** and a **counit**;

1. the **unit** is natural transformation made with universal arrows $\eta\colon I \dot{\to} GF$, where the right adjoint of each $f\colon Fx \to a$ is
$$\varphi f = Gf \circ \eta_x\colon x \to Ga.$$
2. the **counit** is natural transformation made with universal arrows $\varepsilon\colon FG \dot{\to} I$, where the left adjoint of each $g\colon x \to Ga$ is
$$\varphi^{-1} g = \varepsilon \circ Fg\colon Fx \to a.$$

that follow the *triangle identities* $\eta G \circ G\varepsilon = \mathrm{id}$ and $F\eta \circ \varepsilon F = \mathrm{id}$.

Proposition 11 (Characterization of adjunctions)**.** *Each adjunction is completely determined by any of*

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 $\eta_x\colon x \to GF_0x$, creating a functor F.*
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 units and counits satisfiying the triangle identities $\eta G \circ G\varepsilon = \mathrm{id}$ and $F\eta \circ \varepsilon F = \mathrm{id}$.*

Proof. □

16

MONADS AND ALGEBRAS

16.1 MONADS

Definition 60 (Monad)**.** A **monad** is a functor $T\colon X \to X$ with natural transformations

- $\eta\colon I \dot{\to} T$, called *unit*
- $\mu\colon T^2 \dot{\to} T$, called *multiplication*

such that

$$\begin{array}{ccc} T^3 & \xrightarrow{T\mu} & T^2 \\ \downarrow{\scriptstyle \mu T} & & \downarrow{\scriptstyle \mu} \\ T^2 & \xrightarrow{\mu} & T \end{array} \qquad \begin{array}{ccccc} IT & \xrightarrow{\eta T} & T^2 & \xleftarrow{T\eta} & TI \\ & \searrow^{\cong} & \downarrow{\scriptstyle \mu} & \swarrow_{\cong} & \\ & & T & & \end{array} .$$

Proposition 12 (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

$$\begin{array}{ccc} FGFG & \xrightarrow{FG\varepsilon} & FG \\ {\scriptstyle \varepsilon FG}\downarrow & & \downarrow{\scriptstyle \varepsilon} \\ FG & \xrightarrow{\varepsilon} & I \end{array} \qquad \begin{array}{ccc} GFGFGF & \xrightarrow{GFG\varepsilon F} & GFGF \\ {\scriptstyle G\varepsilon FGF}\downarrow & & \downarrow{\scriptstyle G\varepsilon F} \\ GFGF & \xrightarrow{G\varepsilon F} & GF \end{array} ,$$

where the first is commutative by the [BROKEN LINK: *Interchange law] 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. □

16.2 COMONADS

Definition 61 (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

$$\begin{array}{ccc} L & \xrightarrow{\delta} & L^2 \\ {\scriptstyle\delta}\downarrow & & \downarrow{\scriptstyle L\delta} \\ L^2 & \xrightarrow{\delta L} & L^3 \end{array} \qquad \begin{array}{ccccc} & & L & & \\ & \swarrow^{\cong} & \downarrow{\scriptstyle\delta} & \searrow^{\cong} & \\ IL & \xleftarrow[\varepsilon L]{} & L^2 & \xrightarrow[L\varepsilon]{} & LI \end{array} .$$

16.3 ALGEBRAS FOR A MONAD

Definition 62 (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

$$\begin{array}{ccc} T^2x & \xrightarrow{Th} & Tx \\ {\scriptstyle\mu}\downarrow & & \downarrow{\scriptstyle h} \\ Tx & \xrightarrow{h} & x \end{array} .$$

Definition 63 (Morphism of T-algebras)**.** A **morphism of T-algebras** is an arrow $f\colon x \to x'$ making the following square commute

$$\begin{array}{ccc} Tx & \xrightarrow{h} & Tx \\ {\scriptstyle Tf}\downarrow & & \downarrow{\scriptstyle f} \\ Tx' & \xrightarrow[h']{} & Tx' \end{array} .$$

Proposition 13 (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

$$\begin{array}{ccc} Tx & \xrightarrow{h} & x \\ {\scriptstyle Tf}\downarrow & & \downarrow{\scriptstyle f} \\ Tx' & \xrightarrow{h'} & x' \\ {\scriptstyle Tg}\downarrow & & \downarrow{\scriptstyle g} \\ Tx'' & \xrightarrow{h''} & x'' \end{array}$$

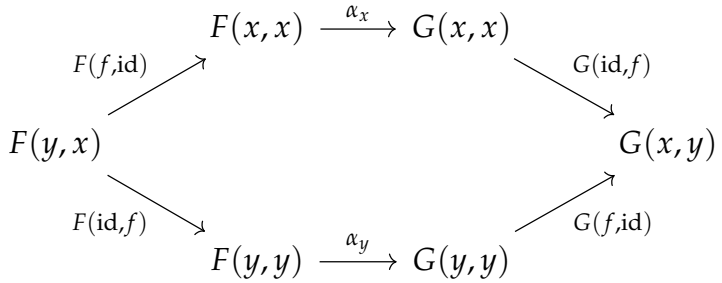
commutes. □

17

KAN EXTENSIONS

17.1 DINATURAL TRANSFORMATIONS

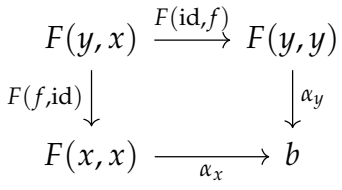
Definition 64 (Dinatural transformation)**.** A **dinatural transformation** between functors $F, G\colon \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{B}$ is a family of morphisms $\alpha_x\colon F(x,x) \to G(x,x)$ for each $x \in \mathcal{C}$, called *components*, such that for every $f\colon x \to y$,



commutes

Definition 65 (Extranatural transformation)**.** An **extranatural transformation** (also called **wedge**) is a dinatural transformation to a constant functor.

That is, an extranatural transformation from the functor F to the object a is a family of morphisms $\alpha_x\colon F(x,x) \to b$ such that



commutes for every $f\colon x \to y$. We write wedges as $\alpha\colon F \dddot{\to} b$.

17.2 ENDS

Definition 66 (End)**.** An **end** of $F\colon \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{B}$ is a universal dinatural transformation from a constant e to F. That is, a wedge $\sigma\colon e \ddot{\to} F$ such that for every other $\tau\colon k \ddot{\to} F$, there exists a unique $u\colon k \to e$ such that $\sigma_x \circ u = \tau_x$.

We usually call *end* to the object e, and we write

$$e = \int_x F(c, c).$$

17.3 COENDS

17.4 KAN EXTENSIONS

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.

18

LAWVERE THEORIES

18.1 MOTIVATION FOR ALGEBRAIC THEORIES

We will develop an unified approach to the study of algebraic structures based on constants, operatio4ns 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 *binary operation* called $\cdot$,
- a *unary operation* written as $^{-1}$, and
- a *nullary operation* or a *constant* called e.

Satisfiying the following equations

- $(x \cdot y) \cdot z = x \cdot (y \cdot z)$,
- $x \cdot e = x$,
- $e \cdot x = x$,
- $x \cdot x^{-1} = e$, and
- $x^{-1} \cdot x = e$.

Quantifiers are not needed here, as we are interpreting each x, y, z as free variables.

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. A theory can be **interpreted** on a 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 $x_1, \ldots, x_n$ if all variables that appear in t appear in $x_1, \ldots, x_n$. We write that as

$$x_1, x_2, \ldots, x_n \mid t;$$

and the **interpretation of the term** $x_1, \ldots, x_n \mid t$ **on that context** is a morphism $I(x_1, \ldots, x_n \mid t) \colon A^n \to A$ defined inductively knowing that

- the interpretation of the i-th variable is the i-th projection

$$I(x_1, \ldots, x_n \mid x_i) = \pi_i : A^n \to A,$$

- the interpretation of an operation over a term is the interpretation of the morphism componsed with the componentwise interpretation of subterms

$$I(f\langle t_1, \ldots, t_k\rangle) = If \circ \langle It_1, \ldots, It_k\rangle \colon A^n \to A.$$

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

$$x \,/\, \big(((x/x)/y)/z\big) \,/\, \big((x/x)/x\big)/z\big) = y$$

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* $[x_1, \ldots, x_n]$ as objects. A morphism from $[x_1, \ldots, x_n]$ to $[x_1, \ldots, x_m]$ will be a tuple of terms

$$\langle t_1, \ldots, t_k\rangle \colon [x_1, \ldots, x_n] \to [x_1, \ldots, x_m]$$

such that every t_k is given in the context $[x_1, \ldots, x_n]$. Composition is defined componentwise as substitution of the terms of the first morphism into the variables of the second one, that is,

$$\langle s_1, \ldots, s_n\rangle = \langle u_1, \ldots, u_n\rangle \circ \langle t_1, \ldots, t_m\rangle,$$

where

$$s_i = u_i[t_1, \ldots, t_m / x_1, \ldots, x_m].$$

Two morphisms in this category $\langle t_1, \ldots, t_n \rangle$ and $\langle s_1, \ldots, s_n \rangle$ 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

$$M[x_1, \ldots, x_n] = A^k$$

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 $\langle x_i \rangle$;
- $M\langle t_1, \ldots, t_m \rangle = \langle Mt_1, \ldots, Mt_m \rangle \colon A^m \to A$, the componentwise interpretation of subterms;
- $M\langle f\langle t_1, \ldots, t_m \rangle \rangle = Mf \circ \langle Mt_1, \ldots, Mt_m \rangle \colon (M\mathbb{A})^m \to M\mathbb{A}$.

The fact that M is a well-defined functor follows from the assumption that it is a model.

18.2 ALGEBRAIC THEORIES AS CATEGORIES

Definition 67 (Lawvere algebraic theory)**.** An **algebraic theory** is a category $\mathbb{A}$ with finite products and objects forming a sequence $A^0, A^1, A^2, \ldots$ 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 68 (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 69 (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 70 (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 13 (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.

18.3 COMPLETENESS FOR ALGEBRAIC THEORIES

Theorem 9 (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 $\mathbb{A}$. □

The universal model needs not to be set-theoretic, but we can always find a universal model in a presheaf category via the Yoneda embedding.

Proposition 14 (Yoneda embedding as a universal model)**.** *The Yoneda embedding $y \colon \mathbb{A} \to \widehat{\mathbb{A}}$ is a universal model for $\mathbb{A}$.*

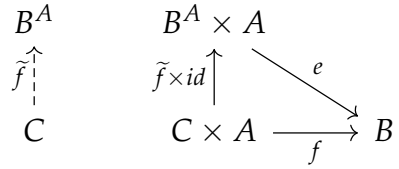
Proof. It preserves finite products because it preserves all limits, hence it is a model. As it is a faithful functor, we know that any equation proved in the model is an eqation proved by the theory. □

19

CARTESIAN CLOSED CATEGORIES

19.1 EXPONENTIAL

Definition 71 (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



An object A for which the exponentiation $-^A$ is always defined is called **exponentiable**.

Proposition 15 (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. □

19.2 CARTESIAN CATEGORY

Definition 72 (Cartesian category)**.** A **cartesian category** is a category with all finite products.

Definition 73 (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 16 (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. □

Definition 74 (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**.

19.3 FRAMES AND LOCALES

Proposition 17 (Completeness for posets)**.** *Complete posets are cocomplete. Cocomplete posets are complete.*

Proof. □

Definition 75 (Frames)**. Frames** are complete cartesian closed posets.

Definition 76 (Frame morphism)**.** A **frame morphism** is a function between frames preserving finite infima and arbitrary suprema.

20

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.

20.1 LATTICES AND BOOLEAN ALGEBRAS

Definition 77 (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 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 78 (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 79 (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 80 (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 18 (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 81 (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.

20.2 HEYTING ALGEBRAS

Definition 82 (Heyting algebra)**.** A **Heyting algebra**, also called **Brouwerian lattice**, is a bounded lattice which is cartesian closed as a category; i.e., 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 83 (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 19 (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 84 (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 20. *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 21 (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 22 (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. □

20.3 INTUITIONISTIC PROPOSITIONAL CALCULUS

Proposition 23 (Existence of free Heyting algebras)**.**

Definition 85 (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 86 (Classical propositional calculus)**.** The **Classical Propositional Calculus** (CPC) is the free Boolean algebra over an infinite countable set $\{p_0, p_1, p_2, \dots\}$.

20.4 QUANTIFIERS AS ADJOINTS

Definition 87. 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 10. *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 11. *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 $\forall_f$.*

21

SIMPLY-TYPED λ-THEORIES

21.1 SIMPLY-TYPED λ-THEORIES

Definition 88 (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.
- a set of *equations* on those terms, of the form $\Gamma \mid u = t : A$, where Γ is the variable context.

21.2 INTERPRETATION OF λ-THEORIES

Definition 89 (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 $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ can be interpreted as the object $$

21.3 SYNTACTIC CATEGORIES

Definition 90 (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 $x : A \mid t : B$

Proposition 24 (Syntactic categories are cartesian closed)**.** *The syntactic category of a λ-theory is cartesian closed.*

Proof. □

Definition 91 (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.

21.4 TRANSLATIONS, CATEGORY OF LAMBDA-THEORIES

Definition 92 (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 93 (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

$$\begin{aligned} \tau(\mathtt{fst}\ t) = \mathtt{fst}\ (\tau t), \qquad & \tau(\mathtt{snd}\ t) = \mathtt{snd}\ (\tau t), \quad \tau\langle u, v\rangle = \langle \tau u, \tau v\rangle, \\ \tau(tu) = (\tau t)(\tau u), \quad & \tau(\lambda x : A.t) = \lambda x : \tau A.\tau t, \end{aligned}$$

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 94 (Category of lambda-theories)**.** The category $\check{}\mathtt{Thr}$ has λ-theories as objects and translations between them as morphisms.

21.5 INTERNAL LANGUAGE OF A CATEGORY

Definition 95 (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

22

TOPOI

22.1 SUBOBJECT CLASSIFIER

Definition 96 (Subobject classifier)**.** A **subobject classifier** is an object Ω with a monomorphism $\text{true}\colon 1 \to \Omega$ such that, for every monomorphism $S \to X$, there exists a unique ϕ such that

$$\begin{array}{ccc} S & \longrightarrow & 1 \\ \downarrow & & \downarrow{\scriptstyle \text{true}} \\ X & \xrightarrow{\phi} & \Omega \end{array}$$

forms a pullback square.

22.2 DEFINITION OF A TOPOS

Definition 97 (Topos)**.** An **elementary topos** (plural *topoi*) is a cartesian closed category with all finite limits and a subobject classifier.

Part V

TYPE THEORY

23

INTUITIONISTIC LOGIC

23.1 CONSTRUCTIVE MATHEMATICS

23.2 AGDA EXAMPLE

In intuitionistic logic, the double negation of the LEM holds for every proposition, that is,

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

- Machine proof

```
id : {A : Set} → A → A
id = λ x → x

data Nat : Set where
 S : Nat → Nat

data ⊥ : Set where

¬ : (A : Set) → Set
¬ A = (A → ⊥)

data _+_(A B : Set) : Set where
 inl : A → A + B
 inr : B → A + B

notnotlem : {A : Set} → ¬ (¬ (A + ¬ A))
notnotlem f = f (inr (λ a → f (inl a)))
```

Part VI

CONCLUSIONS

Part VII

APPENDICES

BIBLIOGRAPHY

[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.

[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.

[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.

[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.

[Lan78] Saunders Mac Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer New York, 1978.

[Lei01] Daan Leijen. Parsec, a fast combinator parser. Technical Report 35, Department of Computer Science, University of Utrecht (RUU), October 2001.

[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.

[Tea] Jupyter Development Team. Jupyter notebooks. a publishing format for reproducible computational workflows.

[Tur37] A. M. Turing. Computability and λ-definability. *The Journal of Symbolic Logic*, 2(4):153–163, 1937.

[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.