UNIVERSIDAD
DE GRANADA

Category Theory and Lambda Calculus

Mario Román García

Trabajo Fin de Grado

Doble Grado en Ingeniería Informática y Matemáticas

Tutores
Pedro A. García-Sánchez
Manuel Bullejos Lorenzo

Facultad de Ciencias
E.T.S. Ingenierías Informática y de Telecomunicación

Granada, a junio de 2018

Contents

Abstract

Chapter 1: The λ-calculus is a collection of systems formalizing the notion of functions. They can be seen as programming languages and formal logics at the same time. We focus on the properties of the untyped λ-calculus and simply typed λ-calculus, and we study their logical interpretation.

Chapter 2: 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. We show how to program and prove statements of intuitionistic propositional logic with it.

Chapter 3: Categories will be the framework we will use to study the fundamental notion of function composition inside mathematics. They provide a useful language to talk about theories, mathematic and logic. In particular, adjoint functors will be one of the key insights we will use to understand and expand the connection between computation and logic we have described so far. This section is based on [Lan78].

Chapter 4: The internal logic of categories is studied. In particular, cartesian closed categories can be used to model lambda calculus, and their structure serves as a basis for locally cartesian closed categories and topoi, which provide a framework for higher-order logics. Dependent type theory arises as the natural language for these richer categories. This section is based on the work by [MM94] and [ML75].

Chapter 5: Type theory provides both a programming language and a foundation of mathematics. The embedding of a category of sets and some classical principles inside constructive logic are studied. We also present and formalize inside a programming language a first result of Homotopy Type Theory.

Sinopsis

Empezamos exponiendo el cálculo lambda de Church como modelo de computación, prestando especial interés a la relación de sus variantes tipadas con la lógica proposicional intuicionista. Desarrollamos un intérprete basado en estas ideas y utilidades para combinarlo con Jupyter Notebook y con Javascript que lo hacen apto para la docencia.

La perspectiva de la teoría de categorías y en especial la noción de Lawvere de las adjunciones como elemento fundacional nos permiten primero formalizar la estructura lógica del cálculo lambda como el lenguaje interno de las categorías cartesianas cerradas e, inmediatamente después, enriquecer esta estructura en categorías localmente cartesianas cerradas hacia la noción de tipos dependientes. Los tipos dependientes nos proveen un marco lógico de orden superior con una interpretación computacional natural. Categorías, lógica y computación aparecen como tres manifestaciones de una misma noción.

La lógica interna de estas categorías es suficientemente rica como para permitirnos desarrollar teorías matemáticas que además pueden expresarse en un lenguaje de programación con un sistema de tipos suficientemente fuerte que sirve como asistente de demostraciones. Para demostrar el potencial de esta perspectiva, usamos el lenguaje de programación Agda para

formalizar los reales positivos con cortes de Dedekind en teoría de tipos de Martin-Löf y proponemos un ejemplo de topología sintética usando el axioma de Univalencia de Voevodsky.

Nuestras referencias principales son la exposición del cálculo lambda de Selinger [Sel13] en la primera parte, los textos de teoría de categorías de MacLane [Lan78] y Awodey [Awo10] en la segunda, y el libro fundacional de la teoría homotópica de tipos [Uni13] en la parte final. El proyecto nLab [nLa18] ha sido útil para encontrar la bibliografía relevante.

What is the primary tool for such summing up of the essence of ongoing mathematics? Algebra! Nodal points in the progress of this kind of research occur when, as in the case with the finite number of axioms for the metacategory of categories, all that we know so far can be expressed in a single sort of algebra.

– **F. William Lawvere**

Chapter 1

Lambda calculus

1.1 Untyped λ-calculus

When are two functions equal? Classically in mathematics, *functions are graphs*. A function from a domain to a codomain, $f\colon X \to Y$, is seen as a subset of the product space: $f \subset X \times Y$. Any two functions are identical if they map equal inputs to equal outputs; and a function is completely determined by what its outputs are under different inputs. This vision is called ***extensional*** (see [Sel13], where this difference is detailed).

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

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

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

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

would exist, thus contradicting the **regularity axiom** of Zermelo-Fraenkel set theory (see, for example, [Kun11]). In constrast, untyped λ-calculus presents some problems that would never appear in our usual foundations such as non-terminating functions. An approach to solving these is presented in Section 1.2.

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

1.1.1 Untyped λ-calculus

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

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

The use of λ-notation also eases the writing of **higher-order functions**, functions whose arguments or outputs are functions themselves. For instance, $\lambda f.(\lambda y.f(f(y)))$ would be a function taking f as an argument and returning $\lambda y.f(f(y))$, which is itself a function; most commonly written as $f \circ f$. In particular, the following expression

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

evaluates to 6. It can be read as applying the polynomial $x^2 + x$ twice to the initial argument 1.

Definition 1.1. λ-terms are constructed using the following rules:

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

Equivalently, they are given by the Backus-Naur form $\mathsf{Term} ::= x \mid (\mathsf{Term}\ \mathsf{Term}) \mid (\lambda x.\mathsf{Term})$, where x represents any variable.

By convention, we omit outermost parentheses and assume left-associativity, for example, MNP will always mean $(MN)P$. Note that the application of λ-terms is not the same as composition of functions, which is associative. We also consider λ-abstraction as having the lowest precedence. For example, $\lambda x.MN$ should be read as $\lambda x.(MN)$ instead of $(\lambda x.M)N$.

1.1.2 Free and bound variables, substitution

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

Definition 1.2 (Free variables)**.** Any ocurrence of a variable x inside the *scope* of a lambda is said to be ***bound***; and any variable without bound ocurrences is said to be ***free***. The **set**

of free variables of a term M is defined inductively on the structure of lambda terms as

$$\begin{array}{ll} \mathrm{FV}(x) = \{x\}, & \text{for any variable } x, \\ \mathrm{FV}(MN) = \mathrm{FV}(M) \cup \mathrm{FV}(N), & \text{for any two terms } M \text{ and } N, \\ \mathrm{FV}(\lambda x.M) = \mathrm{FV}(M) \setminus \{x\}, & \text{for any variable } x \text{ abstracted over a term } M. \end{array}$$

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 evaluating function applications. For instance, in the previous example, we can evaluate $(\lambda x.\ x^2 + x)(2)$ into 6 by substituting 2 in the place of x inside $x^2 + x$; as in

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

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

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

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

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

Definition 1.3. The **substitution** of a variable x by a term N on M is written as $M[N/x]$ and defined inductively on the structure of lambda terms as

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

where, in the last clause, z is a fresh variable that is not used anywhere inside any of the other expressions.

We could define a criterion for choosing exactly what this new variable should be, or simply accept that this procedure is not *well-defined*, but only *well-defined up to a change on the name of the variables*. This equivalence relation between terms with the same structure but different variable names is 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.

1.1.3 Alpha equivalence

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, $(\lambda x.\lambda y.x\ y)$ is α-equivalent to $(\lambda f.\lambda x.f\ x)$.

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

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

Definition 1.4. α-equivalence is the smallest relation $=_\alpha$ on λ-terms that is both an equivalence relation, that is,

- 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 compatible with the structure of lambda terms, that is,

- 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, $\lambda x.M =_\alpha \lambda y.M[y/x]$.

The last clause captures the idea of freely substituting unused variables inside an expression.

1.1.4 Beta reduction

The core notion of evaluation in λ-calculus is captured by the idea 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 consider first a *one-step reduction* relationship, called $\to_\beta$, and we extend it later 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 that canonical form in general.

Definition 1.5. Single-step β-reduction is the smallest relation on λ-terms capturing the notion of evaluation and preserving the structure of λ-abstractions and applications. That is, the smallest relation containing

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

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

1.1.5 Eta reduction

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

Definition 1.6. η-reduction is the smallest relation on λ-terms satisfiying

- $\lambda x.Mx \to_\eta M$, for any $x \notin \mathrm{FV}(M)$,
- $MN \to_\eta M'N$ and $NM \to_\eta NM'$ for any M, M' such that $M \to_\eta M'$, and
- $\lambda x.M \to_\eta \lambda x.M'$, for any M, M' such that $M \to_\eta M'$.

Note that, in the particular case where M is itself a λ-abstraction, η-reduction is simply a particular case of β-reduction. We define single-step $\beta\eta$-reduction as the union of β-reduction and η-reduction. This relation is written as $\to_{\beta\eta}$ and its reflexive transitive closure is written as $\twoheadrightarrow_{\beta\eta}$.

1.1.6 Confluence

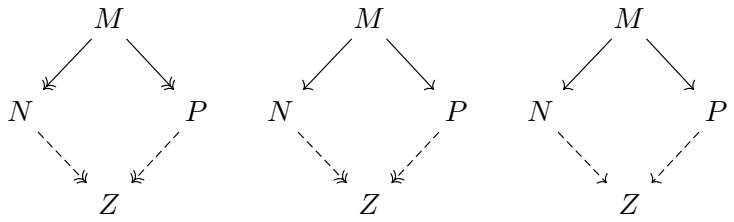
It is not possible in general to evaluate a λ-term into a canonical, non-reducible term. We discuss many examples of this phenomenon in the following sections. However, we will be able to prove that, in the cases where it exists, it is unique. This property is a consequence of a sightly more general one called ***confluence***, which can be defined in any abstract rewriting system.

Definition 1.7 (Confluence)**.** A relation $\to$ on a set $\mathcal{S}$ is **confluent** if, given its reflexive transitive closure $\twoheadrightarrow$ and any terms $M, N, P \in \mathcal{S}$, the relations $M \twoheadrightarrow N$ and $M \twoheadrightarrow P$ imply the existence of some $Z \in \mathcal{S}$ such that $N \twoheadrightarrow Z$ and $P \twoheadrightarrow Z$.

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

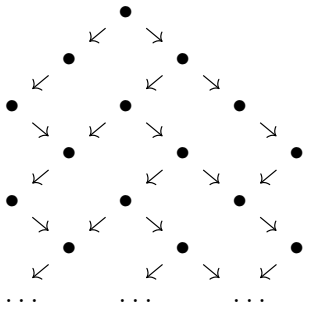
- the **confluence property** (also called *Church-Rosser property*) we have just defined;
- the **quasidiamond property**, similar to the confluence property but assuming $M \to N$ and $M \to P$ instead of the weaker hypothesis of $M \twoheadrightarrow N$ and $M \twoheadrightarrow P$;
- and the **diamond property**, which is defined by substituting $\twoheadrightarrow$ by $\to$ in the definition of confluence.

The three properties can be represented respectively as follows (*left:* confluence, *center:* quasidiamond property, *right:* diamond property).



We can show that the diamond relation implies confluence; while the quasidiamond does not. In fact, the following figure provides a relation satisfying the quasidiamond property but not

the confluence property (from [Sel13]). If want to prove confluence for a given relation, we must use the diamond property instead of the quasidiamond property.



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

1.1.7 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 1.8. We define the **parallel one-step reduction** relation on λ-terms, $\triangleright$, as the smallest relation satisfying that the following properties hold for any variable x and any terms N, N', M, M' such that $M \triangleright M'$ and $N \triangleright N'$:

- reflexivity for variables, $x \triangleright x$;
- parallel application, $MN \triangleright M'N'$;
- λ-abstraction congruence, $\lambda x.N \triangleright \lambda x.N'$;
- parallel substitution, $(\lambda x.M)N \triangleright M'[N'/x]$;
- and extensionality, $\lambda x.Mx \triangleright M'$, if $x \notin \mathrm{FV}(M)$.

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

Lemma 1.9. *The reflexive transitive closure of $\triangleright$ is $\twoheadrightarrow_{\beta\eta}$. In particular, given any λ-terms P, P',*

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

Proof. In both cases, we apply induction on the structure of the derivation.

1. All possible ways in which we can arrive at $P \to_{\beta\eta} P'$ imply $P \triangleright P'$. They 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;

- $\lambda x.Mx \to M$ with $x \notin \mathrm{FV}(M)$; where we can apply extensionality for $\rhd$ and reflexivity.

2. All the possible ways in which we can deduce $M \rhd M'$ imply $M \to_{\beta\eta} M'$. They are
   - the trivial one, reflexivity;
   - parallel application $NM \rhd N'M'$, where, by induction, we have $M \twoheadrightarrow M'$ and $N \twoheadrightarrow N'$. Using two steps, $NM \twoheadrightarrow N'M \twoheadrightarrow N'M'$ we prove $NM \twoheadrightarrow N'M'$;
   - congruence to λ-abstraction $\lambda x.N \rhd \lambda x.N'$, where, by induction, we know that $N \twoheadrightarrow N'$, so $\lambda x.N \twoheadrightarrow \lambda x.N'$;
   - parallel substitution, $(\lambda x.M)N \rhd M'[N'/x]$, where, by induction, we know that $M \twoheadrightarrow M'$ and $N \twoheadrightarrow N'$. Using multiple steps, $(\lambda x.M)N \twoheadrightarrow (\lambda x.M')N \twoheadrightarrow (\lambda x.M')N' \to M'[N'/x]$;
   - extensionality, $\lambda x.Mx \rhd M'$, where by induction $M \twoheadrightarrow M'$, and trivially, $\lambda x.Mx \twoheadrightarrow \lambda x.M'x$.

Because of this, the reflexive transitive closure of $\rhd$ is a subset and a superset of $\twoheadrightarrow$ at the same time. It follows that they must be equal. □

In order to prove that this newly defined relation has the diamond property, we will define a reduction of a term with the property that it can be reached from any of its parallel one-step reductions. We first prove a lemma on substitution that will handle later the more challenging cases of the proof.

Lemma 1.10 (Substitution Lemma)**.** *Let M, M', U and U' be four lambda terms such that $M \rhd M'$ and $U \rhd U'$. Then, we have $M[U/y] \rhd M'[U'/y]$ for any variable y.*

Proof. By structural induction on the derivations of $M \rhd M'$ we have the following cases, depending on what was the last derivation rule we used. Note that we are implicitly assuming the Barendregt's variable convention: all variables have been renamed to avoid clashes.

- Reflexivity, $M = x$. If x precisely is the variable we are substituting, with $x = y$, we simply use $U \rhd U'$; if not, $x \neq y$, we use reflexivity on x to get $x \rhd x$.
- Parallel application. Let M and M' be PN and $P'N'$ respectively, where $P \rhd P'$ and $N \rhd N'$. By induction hypothesis $P[U/y] \rhd P'[U'/y]$ and $N[U/y] \rhd N'[U'/y]$, hence $(PN)[U/y] \rhd (P'N')[U'/y]$ by definition of substitution.
- Congruence. By induction, $N[U/y] \rhd N'[U'/y]$ and therefore $\lambda x.N[U/y] \rhd \lambda x.N'[U'/y]$.
- Parallel substitution. Let M and M' be $(\lambda x.P)N$ and $(\lambda x.P')N'$ respectively, where $P \rhd P'$ and $N \rhd N'$. By induction hypothesis, $P[U/y] \rhd P'[U'/y]$ and $N[U/y] \rhd N[U'/y]$, hence $((\lambda x.P)N)[U/y] \rhd P'[U'/y][N'[U'/y]/x] = P'[N'/x][U'/y]$.
- Extensionality. Let M and M' be $\lambda x.Px$ and $\lambda x.P'x$ respectively, where $x \notin \mathrm{FV}(P)$. By induction hypothesis, $P \rhd P'$, hence $\lambda x.P[U/y]x \rhd P'[U'/y]$. □

Definition 1.11. The **maximal parallel one-step reduct** M^* of a λ-term M is defined inductively as

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

Lemma 1.12 (Diamond property of parallel reduction)**.** *Given any M' such that $M \rhd M'$, it can be proved that $M' \rhd M^*$. Parallel one-step reduction has the diamond property.*

Proof. We apply again induction on the possible derivations 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 1.10) 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 1.13 (Church-Rosser Theorem)**.** *The relation $\twoheadrightarrow_{\beta\eta}$ is confluent.*

Proof. (Tait, Martin-Löf). Parallel reduction, $\rhd$, satisfies the diamond property (Lemma 1.12), which implies the Church-Rosser property. Its reflexive transitive closure is $\twoheadrightarrow_{\beta\eta}$ (Lemma 1.9), whose diamond property implies confluence for $\to_{\beta\eta}$. □

1.1.8 Normalization

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

Definition 1.14. 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 means to iteratively apply reductions to them until a normal form is reached. We know, by virtue of Theorem 1.13, that if a normal form for a particular term exists, then it is unique; but we do not know whether a normal form actually exists. We say that a term *has a normal form* if it can be reduced to a normal form.

Definition 1.15. A term is *weakly normalizing* if there exists a sequence of reductions from it to a normal form. A term is *strongly normalizing* if every possible sequence of reductions is finite.

A consequence of Theorem 1.13 is that a weakly normalizing term has a unique normal form. Strong normalization implies weak normalization, but the converse is not true; as an example, the term $\Omega = (\lambda x.(xx))(\lambda x.(xx))$ is neither weakly nor strongly normalizing; and the term $(\lambda x.\lambda y.y)\ \Omega\ (\lambda x.x)$ is weakly but not strongly normalizing. It can be reduced to a normal form as

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

1.1.9 Standarization and evaluation strategies

We would like to find a β-reduction strategy such that, if a term has a normal form, it can be found by following that strategy. Our basic result will be the **standarization theorem**, which shows that, if a β-reduction to a normal form exists, then a sequence of β-reductions from left to right on the λ-expression will be able to find it. From this result, we will be able to prove that the reduction strategy that always reduces the leftmost β-abstraction will always find a normal form if it exists. This section follows [Kas00], [Bar94] and [Bar84].

Definition 1.16. Any two lambda terms M and N are related by $\to_n$, and we write this as $M \to_n N$, when N can be obtained by β-reducing the n-th leftmost β-reducible application in the lambda term M. We call $\to_1$ the **leftmost one-step reduction** and we write it as $\to_l$; accordingly, $\twoheadrightarrow_l$ is its reflexive transitive closure.

Definition 1.17 (Standard sequence)**.** Let $M_0, M_1, \ldots, M_k$ be a sequence of lambda terms. A sequence of 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 $n_0 \leq n_1 \leq \cdots \leq n_k$, that is, $\{n_i\}$ 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. This will imply the existence of an optimal beta reduction strategy that will always reach a normal form if one exists.

Theorem 1.18 (Standarization theorem)**.** *If $M \twoheadrightarrow_\beta N$, there exists a standard sequence from M to N.*

Proof. (Kashima, 2000) We start by defining the following two binary relations. The first one is the minimal reflexive transitive relation on λ-terms capturing a form of β-reduction called *head β-reduction*; that is, it is the minimal relation $\twoheadrightarrow_h$ such that

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

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

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

We can check the following trivial properties by structural induction

1. $\twoheadrightarrow_h$ implies $\twoheadrightarrow_l$,
2. $\twoheadrightarrow_s$ implies the existence of a standard β-reduction,
3. $\twoheadrightarrow_s$ is reflexive, by induction on the structure of a term,

4. if $M \twoheadrightarrow_h N$, then $MP \twoheadrightarrow_h NP$,
5. if $M \twoheadrightarrow_h N \twoheadrightarrow_s P$, then $M \twoheadrightarrow_s P$,
6. if $M \twoheadrightarrow_h N$, then $M[P/x] \twoheadrightarrow_h N[P/x]$,
7. if $M \twoheadrightarrow_s N$ and $P \twoheadrightarrow_s Q$, then $M[P/z] \twoheadrightarrow_s N[Q/z]$.

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

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

which, by the third clause (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 1.19 (Leftmost reduction theorem)**.** *We define the* ***leftmost reduction strategy*** *as the strategy that reduces the leftmost β-reducible application at each step. If M has a normal form, the leftmost reduction strategy will lead to it.*

Proof. Note that, if $M \to_n N$, where N is in β-normal form; n must be exactly 1. If M has a normal form and $M \twoheadrightarrow_\beta N$, by Theorem 1.18, 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 $\to_l$. □

1.1.10 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 Backus-Naus form

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

where x can represent any free variable.

Definition 1.20. The **Lambda-transform** of a SKI-term is a λ-term defined recursively as

- $\mathfrak{L}(x) = x$, for any variable x;
- $\mathfrak{L}(I) = (\lambda x.x)$;
- $\mathfrak{L}(K) = (\lambda x.\lambda y.x)$;
- $\mathfrak{L}(S) = (\lambda x.\lambda y.\lambda z.xz(yz))$;
- $\mathfrak{L}(XY) = \mathfrak{L}(X)\mathfrak{L}(Y)$, where X and Y are any two SKI-terms.

Definition 1.21. Before translating back lambda terms to SKI combinators, we need the auxiliary notion of bracket abstraction. The **bracket abstraction** of the SKI-term W on the variable x is written as $[x].W$ and defined recursively as

- $[x].x = I$;
- $[x].U = KU$, if $x \notin \mathrm{FV}(U)$;
- $[x].Vx = V$, if V is a term such that $x \notin \mathrm{FV}(V)$;
- $[x].VV' = S([x].V)([x].V')$, otherwise.

where FV is the set of free variables as in Definition 1.2.

Definition 1.22 (SKI abstraction)**.** The **SKI abstraction** of a λ-term P, written as $\mathfrak{H}(P)$ 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 1.23 (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, we have that $\mathfrak{H}(\mathfrak{L}(U)) = U$.*

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. For instance, $\mathfrak{L}(\mathfrak{H}(\lambda u.vu)) = \mathfrak{L}(v) = v$. However, the λ-terms can be essentially recovered if we relax equality between λ-terms to mean $=_{\beta\eta}$.

Theorem 1.24 (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, we know that $\mathfrak{L}([x].M) = \mathfrak{L}(KM) = (\lambda x.\lambda y.x)\mathfrak{L}(M) =_\beta \lambda x.\mathfrak{L}(M)$; and finally, if x is free in U, we have that $\mathfrak{L}([x].Ux) = \mathfrak{L}(U) =_\eta \lambda x.\mathfrak{L}(U)x$. Now we can use this result to prove the main theorem. Again by structural induction,

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

1.1.11 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 unbounded 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 computed using *Turing machines*, theoretical models of a machine.

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 Chapter 2.1; and a detailed description on how to use the untyped λ-calculus as a programming language is given in Chapter 2.5. General recursive functions, for example, can be encoded using these techniques, thus proving that it is in fact Turing complete (see 2.5.7). Note that the lambda calculus has even a cost model allowing us to develop complexity theory within it (see [DLM08]). It is however beyond the scope of this text.

1.2 Simply typed λ-calculus

Types were introduced in mathematics as a response to the Russell's paradox, found in the first naive axiomatizations of set theory. An attempt to use untyped λ-calculus as a foundational logical system by Church suffered from the ***Rosser-Kleene paradox*** and types were a method to avoid it, as detailed in [KR35] and [Cur46]. Once types are added to the calculus, a deep connection between λ-calculus and logic arises. This connection will be discussed in Section 1.3.

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

Simply typed λ-calculus is a refinement of the untyped λ-calculus. On it, each term has a type, which limits how it can be combined with other terms. Only a set of basic types and function types between any to types are considered in this system. Whereas functions in untyped λ-calculus could be applied over any term, once we introduce types 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. Note that A and B can be, themselves, function types.

We present now an account of simply typed λ-calculus based on [HS08]. Our description 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 is clearer to present first a minimal version of the λ-calculus. Such extensions will be explained later, and its exposition will profit from the logical interpretation that we develop in Section 1.3.3.

1.2.1 Simple types

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

Definition 1.25 (Simple types)**.** The set of **simple types** is generated by the Backus-Naur form $\mathsf{Type} ::= \iota \mid \mathsf{Type} \to \mathsf{Type}$, where ι can 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.

1.2.2 Typing rules for simply typed λ-calculus

We define the terms of simply typed λ-calculus using the same constructors we used on the untyped version. The set of **typed lambda terms** is given by the following Backus-Naus form.

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

The main difference here with Definition 1.1 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 $(\lambda x^A.m)(\lambda x^A.m)$. In particular, we can not apply a function of type $A \to B$ to a term of type $A \to B$; because it would expect only a term of type A. **Typing rules** will give terms their desired expressive power. Only a subset of these raw lambda terms can be obtained using the rules, and we will choose to work only with that subset. When a particular term m has type A, we write this relation as $m : A$. The : symbol should be read as *"is of type"*.

Definition 1.26 (Typing context)**.** A **typing context** is a sequence of type assumptions $\Gamma = (x_1 : A_1, \ldots, x_n : A_n)$, where no variable x_i appears more than once. We will implicitly 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 that $x : A$ can be written as $\Gamma, x : A$; and we can trivially deduce from it that $x : A$.

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

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 an *introduction* 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 an *elimination* of function terms.

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

A term m is **typeable** in a giving context Γ if a typing judgement of the form $\Gamma \vdash m : T$ can be derived using only the previous typing rules. From now on, we only consider typeable terms as the terms of simply typed λ-calculus: the set of λ-terms of simply typed λ-calculus is only a subset of the terms of untyped λ-calculus.

Example 1.27 (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, as it can be shown by structural induction on the term ψ.

It can be seen that the typing derivation of a term somehow encodes the complete λ-term. If we were to derive the term bottom-up, there would be only one possible choice at each step on which rule to use. In Section 1.2.4 we will discuss a type inference algorithm that determines if a type is typeable and what its type should be, and we will make precise this intuition.

1.2.3 Curry-style types

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

- **Church-style** typing, also known as *explicit typing*, originated from the work of Alonzo Church in [Chu40], where he described a simply-typed lambda calculus with two basic

types. The term's type is defined as an intrinsic property of the term; and the same term has to be always interpreted with the same type.
- **Curry-style** typing, also known as *implicit typing*, creates a formalism where every single term can be given an infinite number of possible types. This technique is called *polymorphism* when it is a formal part of the language; but here, it is only used to allow us to build intermediate terms without having to directly specify their type.

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

Assuming an infinite numerable set of **type variables**, we define **type templates** as inductively generated by $\mathsf{TypeTemp} ::= \iota \mid \mathsf{Tvar} \mid \mathsf{TypeTemp} \to \mathsf{TypeTemp}$, where ι is a basic type and TVar is a type variable. That is, all basic types and type variables are atomic type templates; and we also consider the arrow type between two type templates. The interesting property of type variables is that they can act as placeholders and be substituted for other type templates.

Definition 1.28. A **type substitution** ψ is any function from type variables to type templates. Any type substitution ψ can be extended to a function between type templates called $\overline{\psi}$ and defined inductively by

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

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

We consider a type to be *more general* than other if the latter can be obtained by applying a substitution to the former. In this case, the latter is called an *instance* of the former. For example, $A \to B$ is more general than its instance $(C \to D) \to B$, where A has been substituted by $C \to D$. A crucial property of simply typed λ-calculus is that every type has a most general type, called its *principal type*; this is proved in Theorem 1.31.

Definition 1.29 (Principal type)**.** A closed λ-term M has a **principal type** π if $M : \pi$, and given any typing judgement $M : \tau$, we can obtain τ as an instance of π, that is, $\overline{\sigma}\pi = \tau$.

1.2.4 Unification and type inference

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

A substitution ψ is called an **unifier** of two sequences of type templates $A_1 \ldots, A_n$ and $B_1, \ldots, B_n$ if $\overline{\psi}A_i = \overline{\psi}B_i$ for any $i = 1, \ldots, n$. 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 1.30 (Unification)**.** *If an unifier of* $\{A_1, \ldots, A_n\}$ *and* $\{B_1, \ldots, B_n\}$ *exists, the most general unifier is* $\mathsf{unify}(A_1, \ldots, A_n; B_1, \ldots, B_n)$*, which is partially defined by induction as follows, where* x *is any type variable.*

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

Moreover, the two sequences of types, $A_1 \ldots, A_n$ *and* $B_1, \ldots, B_n$*, have no unifier if and only if* $\mathsf{unify}(A_1, \ldots, A_n; B_1, \ldots, B_n)$ *fails.*

Proof. It is easy to notice by structural induction that, if $\mathsf{unify}(A; B)$ 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, $\mathsf{unify}(A, B)$ 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 $A_1, \ldots, A_n$ and $B_1, \ldots, B_n$; 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 write an algorithm inferring types.

Theorem 1.31 (Type inference)**.** *The function* $\mathsf{typeinfer}(M, B)$*, partially 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. $\mathsf{typeinfer}(x_i : A_i, \Gamma \vdash x_i : B) = \mathsf{unify}(A_i, B)$*;*
2. $\mathsf{typeinfer}(\Gamma \vdash MN : B) = \overline{\varphi} \circ \psi$*, where* $\psi = \mathsf{typeinfer}(\Gamma \vdash M : x \to B)$ *and* $\varphi = \mathsf{typeinfer}(\overline{\psi}\Gamma \vdash N : \overline{\psi}x)$ *for a fresh type variable* x*;*
3. $\mathsf{typeinfer}(\Gamma \vdash \lambda x.M : B) = \overline{\varphi} \circ \psi$ *where* $\psi = \mathsf{unify}(B; z \to z')$ *and* $\varphi = \mathsf{typeinfer}(\overline{\psi}\Gamma, x : \overline{\psi}z \vdash M : \overline{\psi}z')$ *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 1.30, 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 1.30. 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 1.32 (Principal type property)**.** *Every typeable pure λ-term has a principal type.*

Proof. Given a typeable term M, we can compute $\mathsf{typeinfer}(x_1 : A_1, \ldots, x_n : A_n \vdash M : B)$, 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 1.31, the result is the most general type of M if we assume the variables to have the given types. □

1.2.5 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 1.33 (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. We inductively define the set $\mathtt{RED}_T$ of **reducible** terms of type T for basic and arrow types.

- If $t : T$ where T is 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 $u \in \mathtt{RED}_U$.

We prove three properties of reducibility at the same time in order to use mutual induction.

Proposition 1.34 (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* $t \to_\beta t'$*, then* $t' \in \mathtt{RED}_T$*; 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 definitionally; 2. holds by the definition of strong normalization; and 3. follows from the fact that if any one-step β-reduction leads to a strongly normalizing term then 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 1.35 (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 1.34, $(\lambda x.v) \in \mathtt{RED}_{U \to V}$. □

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 1.36 (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 1.35 to get that $\tilde{t} = \lambda y.\tilde{v} \in \mathtt{RED}_{R \to S}$. □

Theorem 1.37 (Strong Normalization Theorem)**.** *In simply typed* λ*-calculus, all terms are strongly normalizing.*

Proof. It is the particular case of Lemma 1.36 where we take $u_i = x_i$. □

Every term normalizes in simply typed λ-calculus and every computation ends, therefore, simply typed λ-calculus must be not Turing complete.

1.3 The Curry-Howard correspondence

1.3.1 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 new syntax will make our λ-calculus more expressive, but the unification and type inference algorithms will continue to work in a similar way. The previous proofs and algorithms can be extended to cover all the new cases.

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 is 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 new set of **typed lambda terms** is given by the BNF

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

And 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) and (app) rules construct and apply function terms.
$$(abs)\ \frac{\Gamma, x : A \vdash m : B}{\Gamma \vdash \lambda x.m : A \to B} \qquad (app)\ \frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B}$$
3. The $(pair)$ rule constructs pairs of elements. The (π_1) and (π_2) rules destruct a pair into its projections.
$$(pair)\ \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \langle a, b \rangle : A \times B} \qquad (\pi_1)\ \frac{\Gamma \vdash m : A \times B}{\Gamma \vdash \pi_1\ m : A} \qquad (\pi_2)\ \frac{\Gamma \vdash m : A \times B}{\Gamma \vdash \pi_2\ m : B}$$
4. The (inl) and (inr) rules provide the two ways of creating a tagged union type, while the $(case)$ rule extracts a term from a union type applying case analysis. Note that we write $[a].n$ and $[b].p$ to explicitly indicate that n and p can depend on a and b, respectively.
$$(inl)\ \frac{\Gamma \vdash a : A}{\Gamma \vdash \text{inl}\ a : A + B} \qquad (inr)\ \frac{\Gamma \vdash b : B}{\Gamma \vdash \text{inr}\ b : A + B}$$
$$(case)\ \frac{\Gamma \vdash m : A + B \qquad \Gamma, a : A \vdash n : C \qquad \Gamma, b : B \vdash p : C}{\Gamma \vdash (\text{case}\ m\ \text{of}\ [a].n;\ [b].p) : C}$$
5. The $(*)$ rule simply creates the only element of type 1.

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

6. The (*abort*) rule extracts a term of any type from the void type. If we reach a void type, we have reached an error, and thus we can throw any typed exception.

$$(abort)\ \frac{\Gamma \vdash m : 0}{\Gamma \vdash \mathrm{abort}_A\ m : A}$$

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, $(\text{case } m \text{ of } [x].n;\ [y].p) \to_\beta n[a/x]$ if m is of the form $m = \text{inl } a$; and $(\text{case } m \text{ of } [x].n;\ [y].p) \to_\beta p[b/y]$ if m is of the form $m = \text{inr } b$.

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

1. Function extensionality, $\lambda x.f\ x \to_\eta f$ for any $f : A \to B$.
2. Definition of product, $\langle \pi_1\ m, \pi_2\ m\rangle \to_\eta m$ for any $m : A \times B$.
3. Uniqueness of unit, $t \to_\eta *$ for any $t : 1$.
4. Case rule, $(\text{case } m \text{ of } [a].p[\text{inl } a/c];\ [b].p[\text{inr } b/c]) \to_\eta p[m/c]$ for any $m : A + B$.

1.3.2 Natural deduction

The natural deduction is a logical system due to Gentzen. We introduce it here following [Sel13] and [Wad15]. Its relationship with the simply-typed lambda calculus will be made explicit in Section 1.3.3.

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

1. Every axiom on the context can be used.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

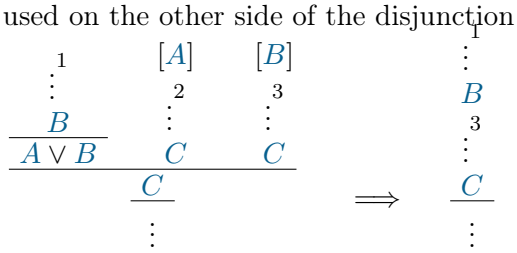
   and, similarly, on the other side as

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

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

$$\dfrac{\dfrac{}{\vdots}^1 A}{A \vee B} \qquad \dfrac{[A]^2 \qquad [B]^3}{\dfrac{}{\vdots} C \qquad \dfrac{}{\vdots} C}{C} \quad\Longrightarrow\quad \dfrac{}{\vdots}^1 A \qquad \dfrac{}{\vdots}^2 C$$

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



1.3.3 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; and
3. simplification of proofs is evaluation of programs.

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

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

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

Now it is easy to notice that every deduction rule of Section 1.3.2 has a correspondence with a typing rule of Section 1.3.1. The only distinction between them is the appearance of λ-terms on the first set of rules. As every typing rule results on the construction of a particular kind of λ-term, they can be interpreted as encodings of proof in the form of derivation trees. That is, terms are proofs of the propositions represented by their types.

Example 1.38 (Curry-Howard correspondence example). In particular, the typing derivation of the term

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

can be seen as a deduction tree proving $A \to B \to A \wedge B$; as the following diagram shows.

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

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

Example 1.39 (Curry-Howard simplification example). As an example of this duality, we will write a proof/term of the proposition/type $A \to B + A$ 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;

$$\dfrac{\dfrac{\dfrac{\dfrac{m : [A + B] \qquad \dfrac{\dfrac{c : A}{\text{inr } c : B + A}\,(inr) \qquad \dfrac{c : B}{\text{inl } c : B + A}\,(inl)}{\text{case } m \text{ of } [c].\text{inr } c; [c].\text{inl } c : B + A}\,(case)}{\lambda m.\text{case } m \text{ of } [c].\text{inr } c; [c].\text{inl } c : A + B \to B + A}\,(abs) \qquad \dfrac{a : A}{\text{inl } a : A + B}\,(inl)}{(\lambda m.\text{case } m \text{ of } [c].\text{inr } c; [c].\text{inl } c)\ (\text{inl } a) : B + A}\,(app)}{\lambda a.((\lambda m.\text{case } m \text{ of } [c].\text{inr } c; [c].\text{inl } c)\ (\text{inl } a)) : A \to B + A}\,(abs)$$

which is encoded by the term $\lambda a.(\lambda m.\text{case } m \text{ of } [c].\text{inr } c; [c].\text{inl } c)\ (\lambda a.\text{inl } a)$. We apply the simplification rule/β-rule of the implication/function application to get

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

which is encoded by the term $\lambda z.\text{case } (\text{inl } z) \text{ of } [a].\text{inr } a; [b].\text{inl } b$. We finally apply the case simplification/reduction rule to get

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

which is encoded by $\lambda a.\text{inr } a$.

On Chapter 2.1, we develop a λ-calculus interpreter which is able to check and simplify proofs in intuitionistic logic. This example could be checked and simplified by this interpreter as it is shown at Figure 1.1.

1.4 Other type systems

1.4.1 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.

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

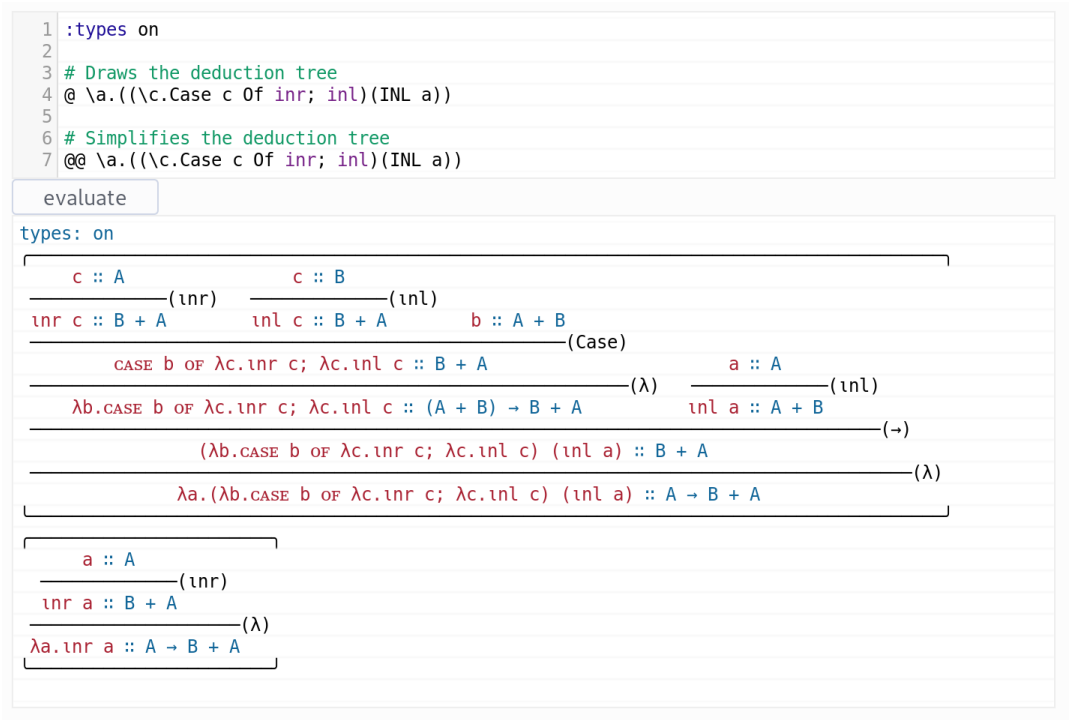


Figure 1.1: Curry-Howard example in Mikrokosmos.

is the following parametric identity function where Λ acts as a λ for types, and τ is a type variable.

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

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

$$\mathrm{id}_A \equiv \lambda x.x : A \to A.$$

2. **Type operators:** types that depend on types. An example of type operator is $[-]$, which sends each type A to the type $[A]$ of lists of elements of A. If we also assume polymorphism, a higher-order function mapping a function argument over a list would have the following type.

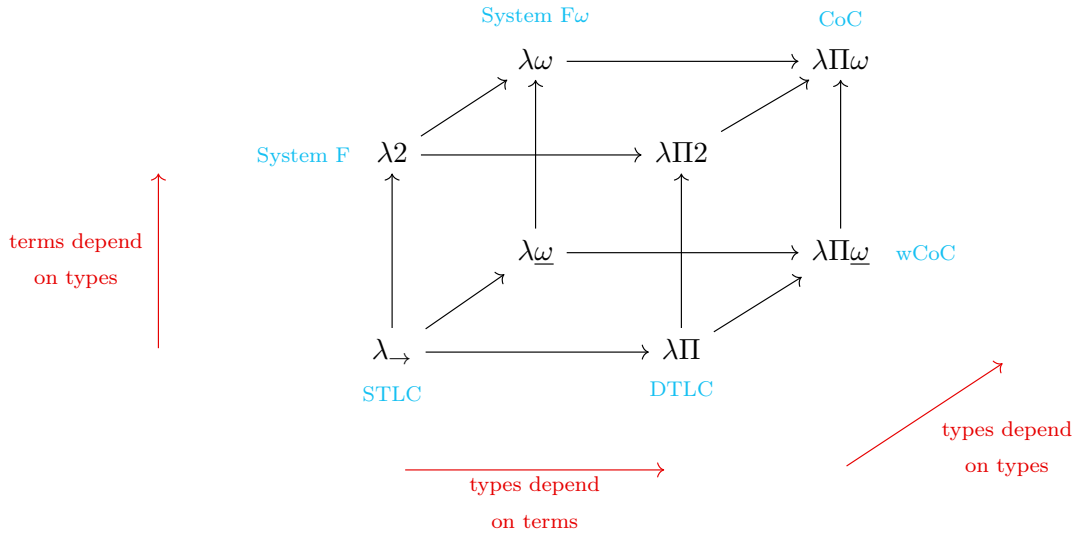
$$\mathrm{map} : \forall\tau.\forall\sigma.(\tau \to \sigma) \to [\tau] \to [\sigma]$$

3. **Dependent types:** types that depend on terms. An example is the type $\mathrm{Vect}(n)$ of vectors of a fixed length n, where n is, itself, an element of a natural numbers type $n : \mathbb{N}$. The type of vectors of any length, $\mathrm{Vect}(0) + \mathrm{Vect}(1) + \mathrm{Vect}(2) + \ldots$, is written as

$$\sum_{n:\mathbb{N}} \mathrm{Vect}(n).$$

Chapters 4.4 and 5.1 are devoted to the study of dependent types.

The λ-cube is shown in the following figure.



It presents the following type systems. Some of which are not commonly used, but all of them are strongly normalizing.

- **Simply typed λ-calculus** ($\lambda_{\rightarrow}$); as described in Section 1.2.2.
- **Simply typed λ-calculus with operators** ($\lambda\underline{\omega}$).
- **System F** ($\lambda 2$) and **System F-omega** ($\lambda\omega$) add polymorphism to the simply typed λ-calculus and type operators. The Haskell programming language is based on System F-omega with some restrictions.
- **Dependently typed λ-calculus** ($\lambda\Pi$); used in the Edinburgh Logical Framework for logic programming (see [HHP93]).
- **Calculus of constructions** ($\lambda\Pi\omega$); where full mathematical theories can be developed (see [CH88]). It is used in the Coq Proof Assistant.

The λ-cube is generalized by the theory of pure type systems, described in [Bar92] and [Geu93].

Chapter 2

Mikrokosmos

2.1 Implementation of λ-expressions

2.1.1 The Haskell programming language

Haskell is the purely functional programming language of our choice to implement our λ-calculus interpreter. Its own design is heavily influenced by the λ-calculus and it 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 and justify our choice.

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, so the **Haskell 98 Report** was a first standarized reference of a common lazy functional language. A revised version can be read in [P+03]. We will use its most standard implementation: 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], but we are interested in the following properties: Haskell is

1. **strongly and statically typed**, meaning that it only compiles well-typed programs and it does not allow implicit type casting; type declarations are then useful in our interpreter to keep a track of what kind of data are we dealing with at each specific function;
2. **lazy**, with *non-strict semantics*, meaning that it will not evaluate a term or the argument of a function until it is needed; this can help to solve the traditional efficiency problems on functional programming (see [Hug89]);
3. **purely functional**; as the evaluation order is demand-driven and not explicitly known, it is not possible to perform ordered input/output actions or any other side-effects that rely on the evaluation order; this helps modularity of our code, testing, and verfication;
4. **referentially transparent**; as a consequence of its purity, every term on the code can be replaced by its definition without changing the global meaning of the program; this allows equational reasoning with rules that are directly derived from λ-calculus and makes it easier to reason about our functions;

[1]: A categorized list of libraries can be found in the central package archive of the Haskell community: https://hackage.haskell.org/packages/

5. 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 to activate extensions that implement dependent types.

Example 2.1 (A first example in Haskell). This example shows the basic syntax and how its type system and its implicit laziness can be used.

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

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

```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 or implementing inductive definitions. Note that the majority of the definitions we discussed in Sections 1.1 and 1.2 are precisely structurally inductive.

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 the following example shows. The second one, because of its simplicity, is usually preferred.

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

A characteristic piece of Haskell are **type classes**, which allow defining common interfaces for different types. In the following example, we define `Monad` as the type class of types with suitably typed `return` and `>>=` 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.

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

Monads 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 concatenate computations with monads that resembles an imperative language. The following example uses the list monad to compute the list of Pythagorean triples.

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

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

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

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

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

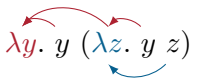
```
roots :: (Float,Float,Float) -> Maybe Int
roots (a,b,c) = do
  -- Some errors can occur during this computation
  discriminant <- sqroot (b*b - 4*c*a)         -- roots of negative numbers?
  root1 <- safeDiv ((-b) + discriminant) (2*a) -- division by zero?
  root2 <- safeDiv ((-b) - discriminant)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

A more detailed treatment of monads from the perspective of category theory is presented in Section 3.5.3.

2.1.2 De Bruijn indexes

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

Consider the following example where we draw arrows between each term and its intermediate λ-abstractions: the λ-term



can be written with de Bruijn indices as $\lambda\ (1\ \lambda\ (2\ 1))$.

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

Definition 2.2 (De Bruijn indexed terms)**.** We define recursively the set of λ-terms using de Bruijn notation as the terms generated from the following Backus normal form.

$$\texttt{Exp} ::= \underbrace{\mathbb{N}}_{variable} \mid \underbrace{(\lambda\ \texttt{Exp})}_{abstraction} \mid \underbrace{(\texttt{Exp}\ \texttt{Exp})}_{application}$$
$$\mathbb{N} ::= 0 \mid 1 \mid 2 \mid \dots$$

Our internal definition closely matches the formal one. The names of the constructors are `Var`, `Lambda` and `App`, representing variables, abstractions and applications, respectively.

```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.

2.1.3 Substitution

We now implement the substition operation described in Section 1.1.2, as it will be needed for β-reducing terms 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 can apply the function `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 an auxiliary function called `incrementFreeVars`; whenever it is applied to a variable, the previous cases are taken into consideration.

```haskell
-- | Substitutes an index for a lambda expression
subs :: Integer -> Exp -> Exp -> Exp
```

```haskell
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
```

Now β-reduction can be defined using this `subs` function.

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

2.1.4 De Bruijn-terms and λ-terms

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

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

The translation from a natural lambda expression to De Bruijn notation is done using a dictionary which keeps track of bounded variables.

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

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

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

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

```haskell
-- | An infinite list of all possible variable names
-- in lexicographical order.
variableNames :: [String]
```

```haskell
variableNames = concatMap (`replicateM` ['a'..'z']) [1..]

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

2.1.5 Evaluation

As we proved on Corollary 1.19, the leftmost reduction strategy will find the normal form of any given term provided that it exists. Consequently, we will implement reduction in our interpreter using a function that simply applies the leftmost possible reductions at each step. As a side benefit, this will allow us to easily show how the interpreter performs step-by-step evaluations to the final user (see Section 2.2.2).

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

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

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

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

2.1.6 Principal type inference

The interpreter implements the *unification* and *type inference* algorithms described in Lemma 1.30 and Theorem 1.31. Their recursive nature makes them very easy to implement directly on Haskell. We implement a simply-typed lambda calculus with *Curry-style typing* (see Section

1.2.3) and type templates. Our type system has a *unit* type; a *void* 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 the whole computation. The algorithm is exactly the same that was defined in Lemma 1.30.

```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
```

```haskell
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. A generalized version of the type inference algorithm is used to generate derivation trees from terms, as it was described in Section 1.3.3. In order to draw these diagrams in Unicode characters, a data type for character blocks has been defined. A monoidal structure is defined over them; blocks can be joined vertically and horizontally; and every deduction step can be drawn independently.

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

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

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

2.2 User interaction

2.2.1 Monadic parser combinators

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

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

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

Given a value, the `return` function creates a parser that consumes no input and simply returns the given value. The `>>=` function acts as a sequencing operator for parsers; it takes two parsers and applies the second one over the remaining inputs of the first one, using the parsed values on the first parsing as arguments.

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

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

An example of **parser combinator** is the `many` function, which creates a parser that allows one or more applications of the given parser. In the following example `many item` would be a parser consuming all characters from the input string.

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

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.

2.2.2 Verbose mode

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

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

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

mikro> :verbose on
verbose: on
mikro> plus 1 2
((plus 1) 2)
((λλλλ((4 2) ((3 2) 1)) λλ(2 1)) λλ(2 (2 1)))
(λλλ((λλ(2 1) 2) ((3 2) 1)) λλ(2 (2 1)))
λλ((λλ(2 1) 2) ((λλ(2 (2 1)) 2) 1))
λλ(λ(3 1) (λ(3 (3 1)) 1))
λλ(2 (λ(3 (3 1)) 1))
λλ(2 (2 (2 1)))
λa.λb.(a (a (a b))) ⇒ 3
```

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

```
-- | 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
```

2.2.3 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 | Cte String
```

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

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

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

```haskell
-- | Bracket abstraction of a SKI term, as defined in Hindley-Seldin
-- (2.18).
bracketabs :: String -> Ski -> Ski
bracketabs x (Cte y) = if x == y then I else Comb K (Cte y)
bracketabs x (Comb u (Cte y))
  | freein x u && x == y = u
  | freein x u           = Comb K (Comb u (Cte y))
  | otherwise            = Comb (Comb S (bracketabs x u)) (bracketabs x (Cte y))
bracketabs x (Comb u v)
  | freein x (Comb u v)  = Comb K (Comb u v)
  | otherwise            = Comb (Comb S (bracketabs x u)) (bracketabs x v)
bracketabs _ a           = Comb K a

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

2.3 Usage

2.3.1 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

```
stack install mikrokosmos                                  # Mikrokosmos interpreter
sudo pip install imikrokosmos                  # Jupyter kernel for Mikrokosmos
git clone https://github.com/mroman42/mikrokosmos-lib.git ~/.mikrokosmos # Libs
```

The **Mikrokosmos interpreter** is listed in the central Haskell package archive[2]. The packaging of Mikrokosmos has been done using the **cabal** tool; and the configuration of the package can be read in the file `mikrokosmos.cabal` of the source code. As a result, Mikrokosmos can be installed using the Haskell package managers **cabal** and **stack**.

```
cabal install mikrokosmos          # Installation with cabal
stack install mikrokosmos          # Installation with stack
```

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

The **Mikrokosmos Jupyter kernel** is listed in the central Python package archive[3]. 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.

```
sudo pip install imikrokosmos      # Installation with pip
```

The installation can be checked by listing the available Jupyter kernels.

```
jupyter kernelspec list            # Checks installation
```

The **Mikrokosmos libraries** can be downloaded directly from their GitHub repository[4]. 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/mroman42/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/mroman42/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
```

2.3.2 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 0.7.0. GNU General Public License Version 3.
mikro> _
```

[3]: The Jupyter-Mikrokosmos package can be found in https://pypi.org/project/imikrokosmos/.
[4]: The repository can be accessed in: https://github.com/mroman42/mikrokosmos-lib.git

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

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

Figure 2.1 is an example session on the mikrokosmos interpreter.

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

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

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

Figure 2.1: Mikrokosmos interpreter session.

2.3.3 Jupyter kernel

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

We have developed a Mikrokosmos kernel for the Jupyter Notebook, allowing the user to write and execute arbitrary Mikrokosmos code on this web application. An example session can be seen on Figure 2.2.

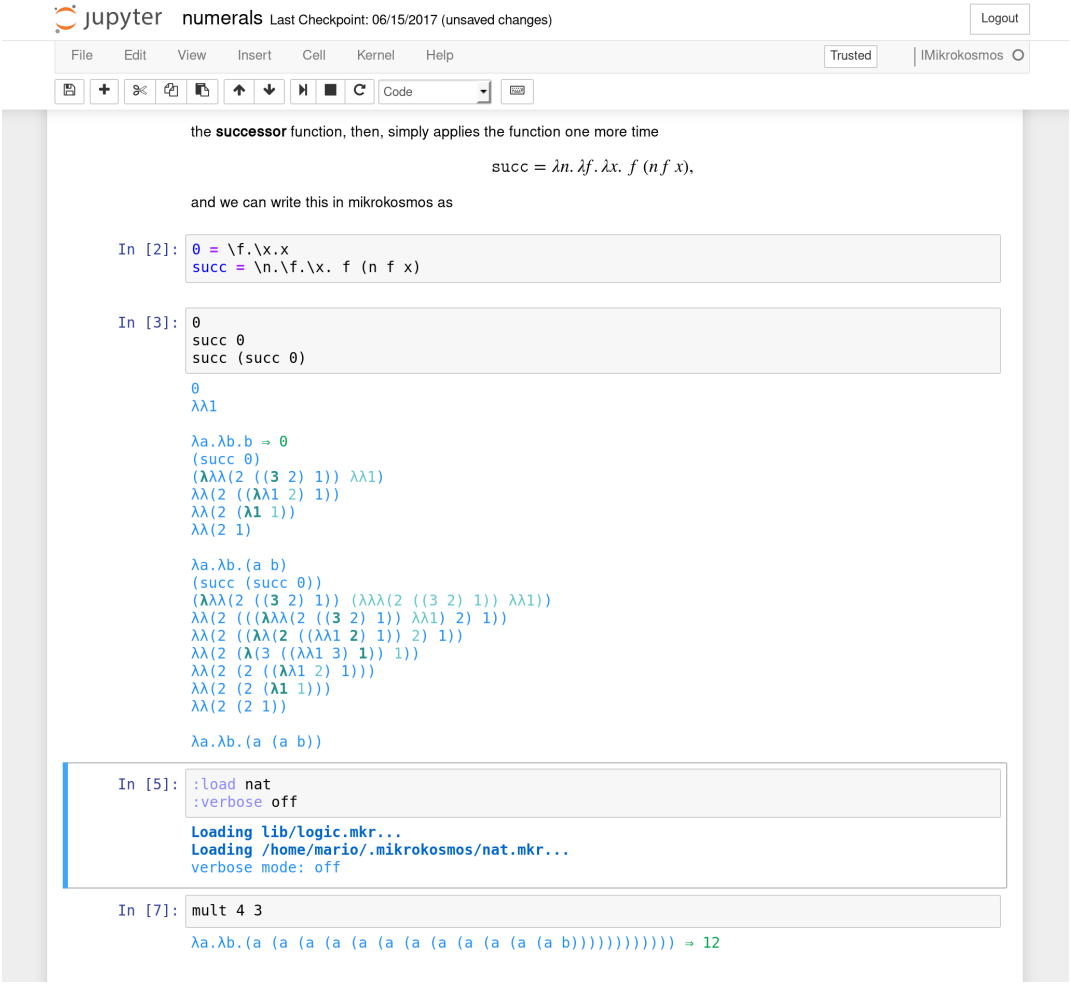


Figure 2.2: Jupyter notebook Mikrokosmos session.

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

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

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

```
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[5]. This allows the kernel to be installed directly using the `pip` python package manager.

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

2.3.4 CodeMirror lexer

CodeMirror [6] is a text editor for the browser implemented in Javascript. It is used internally by the Jupyter Notebook.

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

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

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

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

2.3.5 JupyterHub

JupyterHub manages multiple instances of independent single-user Jupyter notebooks. It has been used to serve Mikrokosmos notebooks and tutorials to students. In order to install Mikrokosmos on a server and use it as `root` user, we perform the following steps.

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

A JupyterHub server was made available at `iemath1.ugr.es`. Our server used a SSL certificate and OAuth autentication via GitHub. Mikrokosmos tutorials and exercises were installed for every student.

2.3.6 Calling Mikrokosmos from Javascript

The GHCjs[7] 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 text as independent Mikrokosmos commands. The relevant use of the foreign function interface is shown in the following code, which provides `mikrokosmos` as a Javascript function once the code is transpiled.

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

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 the following direction, using GitHub as a CDN.

https://mroman42.github.io/mikrokosmos-js/mikrobox.js

The script will convert any HTML script tag written as follows into a CodeMirror pad where Mikrokosmos can be executed.

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

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

```
</script>
</div>
```

The Mikrokosmos tutorials are an example of this feature and can be seen on Figure 2.3. They can be accessed from the following direction.

https://mroman42.github.io/mikrokosmos/

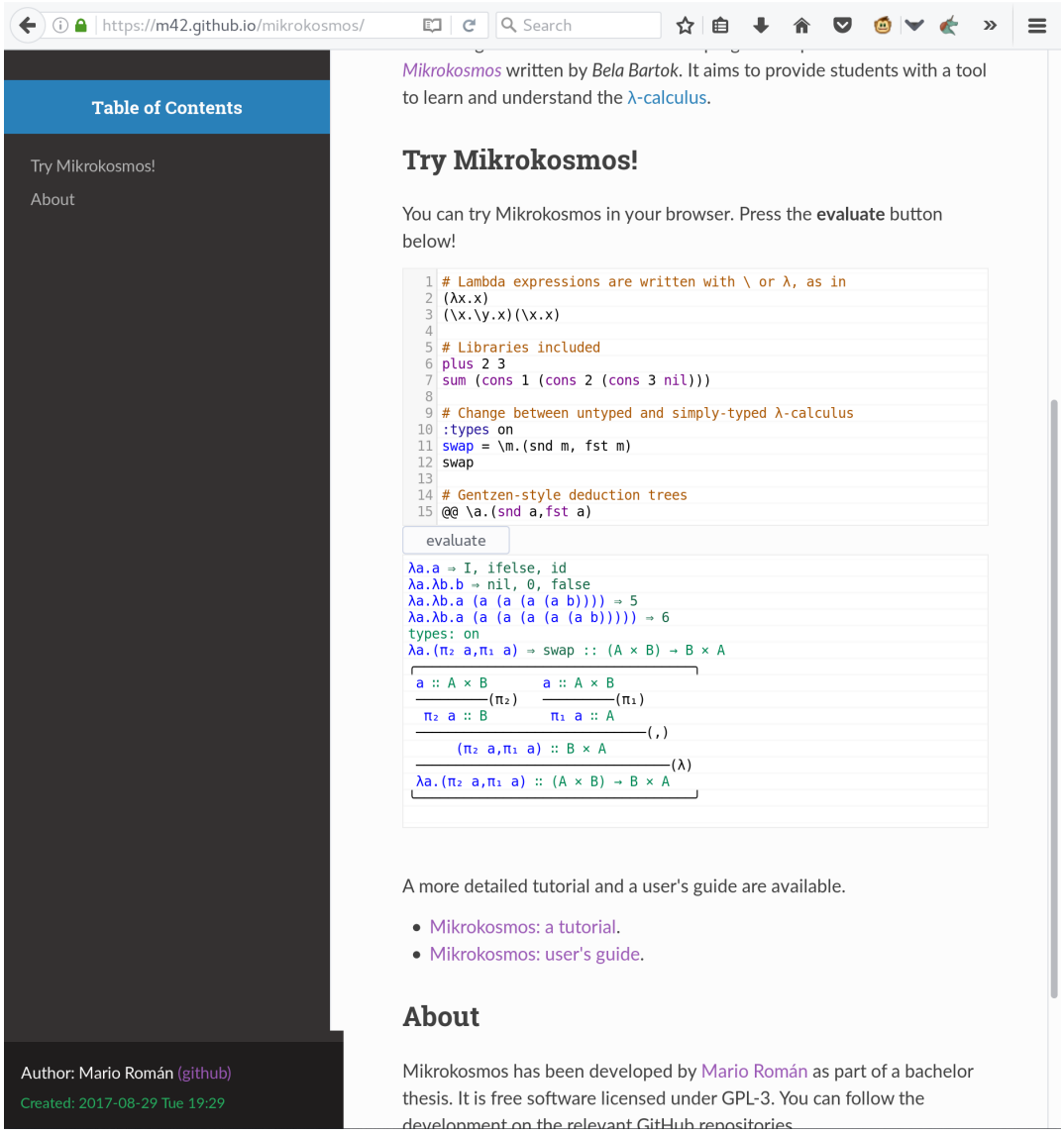


Figure 2.3: Mikrokosmos embedded into a web page.

2.4 Programming environment

2.4.1 Cabal, Stack and Haddock

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

standard for Haskell software.

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

2.4.2 Testing

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

- **unit tests**, in which individual core functions are tested independently of the rest of the application;
- **property-based testing**, in which multiple test cases are created automatically in order to verfiy that a specified property always holds; this has been useful to test our implementation of several algorithms on the lambda calculus;
- **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; they have been used to check correctness of the Mikrokosmos language.

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

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

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

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

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

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

While the expected output is the following.

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

2.4.3 Version control and continuous integration

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

https://github.com/mroman42/mikrokosmos

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

- https://github.com/mroman42/mikrokosmos-js
- https://github.com/mroman42/jupyter-mikrokosmos
- https://github.com/mroman42/mikrokosmos-lib

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

2.5 Programming in untyped λ-calculus

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

This presentation follows the Mikrokosmos tutorial on λ-calculus, which aims to teach how it is possible to program using untyped λ-calculus without discussing more advanced technical topics such as those we addressed on Chapter 1.1. It also follows the exposition on [Sel13] of the usual Church encodings.

All the code on this section is valid Mikrokosmos code.

2.5.1 Basic syntax

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

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

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

Evaluation of terms will be presented as comments to the code, as in the following example.

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

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

```
discard = const id
```

is a function that discards one argument and returns the identity, `id`. This way of defining multiple argument functions is called the **currying** of a function in honor to the american logician Haskell Curry in [CF58]. It is a particular instance of a deeper fact we will detail on Chapter 4.2: exponentials are defined by the adjunction $\hom(A \times B, C) \cong \hom(A, \hom(B, C))$.

2.5.2 A technique on inductive data encoding

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

We start considering the usual inductive representation of a data type with constructors, as we do when representing a syntax with a BNF. For example, the naturals can be written as `Nat` ::= `Zero` | `Succ Nat`; and, in general a datatype D with constructors $C_1, C_2, C_3, \ldots$ is written as `D` ::= C_1 | C_2 | C_3 | ....

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

Our technique, instead, is to define a data term as a function on multiple arguments representing the missing constructors. For instance, the number 2, which would be written as `Succ`(`Succ`(`Zero`)) using the BFN, is encoded as $2 = \lambda s.\ \lambda z.\ s(s(z))$. In general, any instance of the data structure `D` is encoded as a λ-expression depending on all of its constructors, as in $\lambda c_1.\ \lambda c_2.\ \lambda c_3.\ \ldots\ \lambda c_n.(term)$.

This acts as the definition of an initial algebra over the constructors (see Section 3.5.4) and lets us to compute over instances of that algebra by instantiating it on particular cases. We will develop explicit examples of this technique on the following sections.

2.5.3 Booleans

Booleans can be defined as the data generated by a pair of zero-ary constuctors, each one representing a truth value, as in `Bool` ::= `True` | `False`. Consequently, the Church encoding of booleans takes these constructors as arguments and defines the following two elements.

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

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

We can test this interpretation on the interpreter to get the expected results.

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

And this inspires the definition of an `ifelse` combinator as the identity, when applied to boolean values.

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

The usual logic gates can be defined using this interpretation of the 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                                                    --- [1]: false
and true true                                                    --- [2]: true
```

2.5.4 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 in Section 2.5.2.

```
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 follows, even if we can not define an infinite number of terms as we might wish.

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

The interpretation a natural number n as a higher order function is a function taking an argument `f` and applying them n times over the second argument. See the following examples.

```
5 not true                                                    --- [1]: false
4 not true                                                    --- [2]: true
double = \n.\s.\z.n (compose s s) z
double 3                                                      --- [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                                                      --- [1]: 3
mult 2 4                                                      --- [2]: 8
```

2.5.5 The predecessor function and predicates on numbers

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

Stephen Kleene, one of the students of Alonzo Church only discovered how to write the predecessor function after thinking about it for a long time (and he only discovered it while a long visit at the dentist's, which is the reason why this definition is often called the ***wisdom tooth trick***, see [Cro75]). We will use a slightly different version of that definition, as we consider it to be simpler to understand.

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

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

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

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

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

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

2.5.6 Lists and trees

We would need two constructors to represent a list: a `nil` signaling the end of the list and a `cons`, joining an element to the head of the list. For instance, a list would be `cons` 1 (`cons` 2 (`cons` 3 `nil`)). Our definition takes those two constructors into account.

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

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

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

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

They can be defined as follows.

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

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

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

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

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

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

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

2.5.7 Fixed points

A fixpoint combinator is a term representing a higher-order function that, given any function `f`, solves the equation `(x = f x)` for `x`, meaning that, if `(fix f)` is the fixpoint of `f`, the following sequence of equations holds

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

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

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

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

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

Note the use of `iszero` to stop the recursion.

The `fix` function cannot be evaluated without arguments into a closed form, so we have to delay the evaluation of the expression when we bind it using `!=`. Our evaluation strategy, however, will always find a way to reduce the term if it is possible, as we saw in Corollary 1.19; even if it has intermediate irreducible terms.

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

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

```
-- Head of an infinite list of zeroes
head = fold const false
head (fix (cons 0))
-- Compare infinity with other numbers
```

```
infinity := fix succ                                    --- [1]: 0
leq infinity 6                                          --- [2]: false
```

These definitions unfold as

- `fix (cons 0)` = `cons 0 (cons 0 (cons 0 ...))`, an infinite list of zeroes;
- `fix succ` = `succ (succ (succ ...))`, an infinite natural number.

As a final example, we define a minimisation operator that finds the first natural causing a given predicate to return true. This, with all the previous considerations about how to program with natural numbers, can be used to prove that Gödel's μ-recursive functions can be encoded inside untyped lambda calculus, and thus, it is Turing-complete.

```
mu := \p.fix (\f.\n.(p n) n (f (succ n))) 0
mu (\n.eq (mult 2 n) 10)                                --- [1]: 5
```

2.6 Programming in the simply typed λ-calculus

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

This presentation of simply typed structures follows the Mikrokosmos tutorial and the previous sections on simply typed λ-calculus (see Section 1.2). All the code on this section is valid Mikrokosmos code.

2.6.1 Function types and typeable terms

Types can be activated with the commmand `:types on`. If types are activated, the interpreter will infer (see Section 2.1.6) the principal type of every term before its evaluation. The type will then be displayed after the result of the computation.

Example 2.3 (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    --- [1]: λa.a ⇒ id, I, ifelse :: A → A
true  --- [2]: λa.λb.a ⇒ K, true :: A → B → A
false --- [3]: λa.λb.b ⇒ nil, 0, false :: A → B → B
0     --- [4]: λa.λb.b ⇒ nil, 0, false :: A → B → B
1     --- [5]: λa.λb.(a b) ⇒ 1 :: (A → B) → A → B
2     --- [6]: λa.λb.(a (a b)) ⇒ 2 :: (A → A) → A → A
S     --- [7]: λa.λb.λc.((a c) (b c)) ⇒ S :: (A → B → C) → (A → B) → A → C
K     --- [8]: λa.λb.a ⇒ K, true :: A → B → A
```

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

Example 2.4 (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.

2.6.2 Product, union, unit and void types

Until now, we have only used the function type. That is to say that we are working on the implicational fragment of the simply-typed lambda calculus we described when first describing typing rules. We are now going to extend our type system in the same sense we extended (see Section 1.3.1) that simply-typed lambda calculus. The following types are added to the system.

Type	Name	Description
$\to$	Function type	Functions from a type to another
`×`	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`	$\top$	Unital element
`abort`	$\bot \to$ `A`	Empty function
`absurd`	$\bot \to \bot$	Particular empty function

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

Example 2.5 (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                  --- [1]: λa.((SND a),(FST a)) ⇒ swap :: (A × B) → B × A
caseof (inl 1) pred (mult 2) --- [2]: λa.λb.b ⇒ nil, 0, false :: A → B → B
caseof (inr 1) pred (mult 2) --- [3]: λa.λb.(a (a b)) ⇒ 2 :: (A → A) → A → A
\x.((abort x),(absurd x))    --- [4]: λa.((ABORT a),(ABSURD a)) :: ⊥ → A × ⊥
```

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

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

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

2.6.3 A proof in intuitionistic logic

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

Theorem 2.6. *In intuitionistic logic, the double negation of the Law of Excluded Middle holds for every proposition. That is, we know that $\neg\neg(A \vee \neg A)$ for an arbitrary proposition 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 must conclude that $\neg\neg(A \vee \neg A)$. □

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

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

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

whose type is precisely $((\mathtt{A} + (\mathtt{A} \to \bot)) \to \bot) \to \bot$. The derivation tree can be seen directly on the interpreter as Figure 1.1 shows.

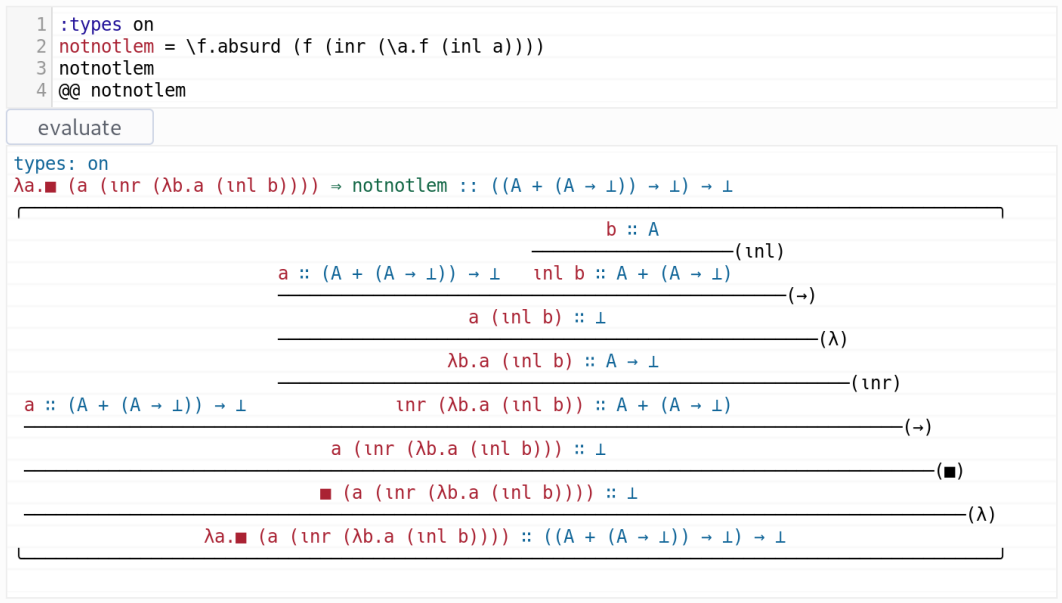


Figure 2.4: Proof of the double negation of LEM.

Chapter 3

Category theory

3.1 Categories

Categories are algebraic structures that capture the notion of composition. They consist of objects linked by composable arrows; to which associativity and identity laws will apply. Thus, a category has to rely in some notion of *collection of objects*. When interpreted inside set-theory, this term can denote sets or proper classes. We want to talk about categories containing subsets of the class of all sets, and thus it is necessary to allow the objects to form a proper class (which is not itself a set) in order to avoid inconsistent results such as the Russell's paradox. This is why we will consider a particular class of categories of small set-theoretical size to be specially well-behaved.

Definition 3.1 (Small and locally small categories)**.** A category is said to be **small** if the collection of its objects can be given by a set (instead of a proper class). It is **locally small** if the collection of arrows between any two objects can be given by a set.

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

3.1.1 Definition of category

Definition 3.2. A **category** $\mathcal{C}$ (following [Lan78] or [Awo10]) is given by a collection whose elements are called *objects*, sometimes denoted $\mathrm{obj}(\mathcal{C})$ or simply $\mathcal{C}$; and a collection whose elements are called *morphisms*. Every morphism f is assigned two objects, its *domain* and its *codomain*; we write $f\colon A \to B$ to indicate that f has domain A and codomain B.

Given any two morphisms $f\colon A \to B$ and $g\colon B \to C$, there exists a **composition** morphism $g \circ f\colon A \to C$. Composition of morphisms is a partially defined binary operation satisfying the following two axioms. It must be *associative*, verifying that $h \circ (g \circ f) = (h \circ g) \circ f$ for any composable morphisms f, g, h; and there must exist an *identity* morphism $\mathrm{id}_A\colon A \to A$ for each object A, verifying that $f \circ \mathrm{id}_A = f = \mathrm{id}_B \circ f$ for all $f\colon A \to B$.

The collection of morphisms between two objects A and B is called an **hom-set** and it is written as $\hom(A, B)$. When necessary, we can use a subscript, as in $\hom_{\mathcal{C}}(A, B)$, to explicitly specify the category we are working in.

3.1.2 Morphisms

Objects in category theory are an atomic concept and can be only studied by their morphisms; that is, by how they relate to each other. In other words, the essence of a category is given not by its objects, but by the morphisms between them and how composition is defined. It is so much so, that we will consider two objects essentially equivalent (and we will call them *isomorphic*) whenever they relate to other objects in the exact same way. This occurs if we can find an invertible morphism connecting both: composition by this morphism or its inverse will translate arrows from one object to the other.

Definition 3.3. A morphism $f : A \to B$ is an **isomorphism** if there exists a morphism $f^{-1} : B \to A$ such that $f^{-1} \circ f = \mathrm{id}_A$ and $f \circ f^{-1} = \mathrm{id}_B$. This morphism is called an **inverse** of f. When an isomorphism between two objects A and B exists, we say they are *isomorphic*, and we write $A \cong B$.

Proposition 3.4. *There is, at most, a single way to invert a morphism. 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 must be equal.*

Proof. Given $f : A \to B$ with a left-side inverse $g_1 : B \to A$ and a right-side inverse $g_2 : B \to A$; we have that

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

As expected, *to be isomorphic to* is an equivalence relation. In particular, reflexivity follows from the fact that the identity is its own inverse, $\mathrm{id} = \mathrm{id}^{-1}$; symmetry follows from the inverse of an isomorphism being itself an isomorphism, $(f^{-1})^{-1} = f$; and transitivity follows by the fact that the composition of isomorphisms is itself an isomorphism, $(f \circ g)^{-1} = g^{-1} \circ f^{-1}$. All these equalities can be checked from the axioms of a category. We can notice that morphisms are an abstraction of the notion of structure-preserving maps between mathematical structures. Two structures can be identified if they are related by an isomorphism. From this perspective, it seems natural to ask how injective and surjective can be described only in terms of composition. *Monomorphisms* and *epimorphisms* will be abstractions of the usual injective and surjective homomorphisms, respectively.

Definition 3.5 (Monomorphisms and epimorphisms)**.** A **monomorphism** is a left-cancellable morphism, that is, $f : A \to B$ is a monomorphism if, for every pair of morphisms $g, h : B \to A$, the equality $f \circ g = f \circ h$ implies $g = h$.

Dually, an **epimorphism** is a right-cancellable morphism, that is, $f : A \to B$ is an epimorphism if, for every $g, h : B \to A$, the equality $g \circ f = h \circ f$ implies $g = h$.

Note that we could have chosen a stronger and non-equivalent notion to generalize the notions of injective and surjective functions.

Definition 3.6 (Retractions and sections)**.** A **retraction** is a left inverse, that is, a morphism that has a right inverse; conversely, a **section** is a right inverse or, in other words, a morphism that has a left inverse.

By virtue of Proposition 3.4, a morphism that is both a retraction and a section is an isomorphism. In general, however, not every epimorphism is a section and not every monomorphism

is a retraction. Consider for instance a category with two objects and a single morphism connecting them; it is a monomorphism and an epimorphism, but it has no inverse. In any case, we will usually work with the more general notion of monomorphisms and epimorphisms.

3.1.3 Products and sums

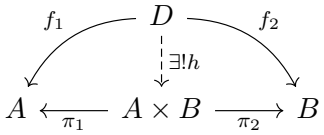
Products and sums are very widespread notions in mathematics. Whenever a new structure is defined, it is common to ask what the product or sum of two of these structures would be. Examples of products are the cartesian product of sets, the product topology or the product of abelian groups; examples of sums are the disjoint union of sets, topological sum or the free product of groups. Following this idea, we can consider certain structures to constitute a 0 or a 1 for these operations; these are called *initial and final objects*.

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

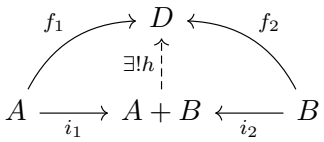
Definition 3.7 (Initial and terminal objects)**.** An object 0 is an **initial object** if for every object A exists an unique morphism of the form $o_A \colon 0 \to A$. An object 1 is a **terminal object** if for every object A exists an unique morphism of the form $*_A \colon A \to 1$.

Note that these objects may not exist in any given category; but when they do, they are essentially unique. If A, B are initial objects, by definition, there are a unique morphism $f : A \to B$, a unique morphism $g : B \to A$, and two unique morphisms $A \to A$ and $B \to B$. By uniqueness, $f \circ g = \mathrm{id}$ and $g \circ f = \mathrm{id}$, hence $A \cong B$. An analogous proof can be written for terminal objects.

Definition 3.8 (Products and sums)**.** An object $A \times B$ with two morphisms $\pi_1 \colon A \times B \to A$ and $\pi_2 \colon A \times B \to B$ is a **product** of A and B if for any other object D with two morphisms $f_1 : D \to A$ and $f_2 : D \to B$, there exists a unique morphism $h : D \to A \times B$, such that $f_1 = \pi_1 \circ h$ and $f_2 = \pi_2 \circ h$ as in the following commutative diagram.



An object $A + B$ with two morphisms $i_1 \colon A \to A + B$ and $i_2 \colon B \to A + B$ is the **sum** of A and B for any other object D with two morphisms $f_1 : A \to D$ and $f_2 : B \to D$, there exists a unique morphism $h : D \to A + B$, such that $f_1 = h \circ i_1$ and $f_2 = h \circ i_2$ as in the following commutative diagram.



Note that neither the product nor the sum of two objects necessarily exist on a category; but when they do, they are essentially unique. This justifies writing them as $A \times B$ and $A + B$. The proof is similar to that of the unicity of initial and terminal objects.

3.1.4 Examples of categories

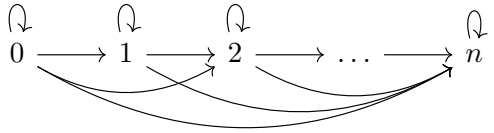
Many mathematical structures, such as sets, groups, or partial orders, are particular cases of a category. Apart from these, we will be also interested in categories whose *objects* are known mathematical structures (the category of all groups, the category of all sets, and so on). The following are examples on how general the definition of a category is.

Example 3.9. A category is **discrete** if it has no other morphisms than the identities. A discrete small category is uniquely defined by the underlying set of its objects and every class of objects defines a discrete category. Thus, small discrete categories can be regarded as sets.

Example 3.10. A single-object category is a **monoid**. A category in which every morphism is an isomorphism is a **groupoid**. A **group** is a category which is a monoid and a groupoid at the same time. These definitions are equivalent to the usual ones if we take morphisms as the elements and composition as the binary operation.

Example 3.11. **Partially ordered sets** are categories with, at most, one morphism between any two objects. We say $a \leq b$ whenever $\rho_{a,b} \colon a \to b$ exists. In a partially ordered set, the product of two objects would be its join, the coproduct would be its meet and the initial and terminal objects would be the greatest and the least element, respectively.

In particular, every ordinal can be seen as a partially ordered set and defines a category. For example, if we take the finite ordinal $[n] = (0 < \cdots < n)$, it could be interpreted as the category given by the following diagram.



Example 3.12 (The category of sets). The category Set is defined as the category with all sets as objects and functions between them as morphisms. It is trivial to check associativity of composition and the existence of the identity function for any set.

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

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

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

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

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

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

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

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

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

Example 3.13 (Groups and modules). The category Grp is defined as the category with groups as objects and group homomorphisms between them as morphisms. 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 3.14 (The category of topological spaces). The category Top is defined as the category with topological spaces as objects and continuous functions between them as morphisms.

3.2 Functors and natural transformations

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

Functors and natural transformations were defined for the first time by Eilenberg and MacLane in [EM42] and [EM45] while studying cohomology theory. While initially they were devised mainly as a language for this study, they have proven its foundational value with the passage of time. The notion of naturality will be a key element of our presentation of algebraic theories and categorical logic.

3.2.1 Functors

Functors can be seen as a homomorphisms of categories that preserve composition and identity arrows. A functor between two categories, $F : \mathcal{C} \to \mathcal{D}$, is given by

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

such that $F(\mathrm{id}) = \mathrm{id}$, and $F(f \circ g) = Ff \circ Fg$. We can arrange functors, at least in the case of small categories, into a category of categories.

Definition 3.15. The category Cat is defined as the category of (small) categories as objects and functors as morphisms.

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

We now consider certain properties of functors in this category. We say that a functor F is **full** if every $g\colon FA \to FB$ is of the form Ff for some morphism $f\colon A \to B$. We say F is **faithful** if, for every two arrows $f_1, f_2\colon A \to B$, $Ff_1 = Ff_2$ 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).

These notions are equivalent to notions of injectivity and surjectivity on the arrow function between any two objects. Note, however, that a faithful functor needs not to be injective on objects nor on morphisms. In particular, if A, A', B, B' are four different objects, it could be the case that $FA = FA'$ and $FB = FB'$; and, if $f : A \to B$ and $f' : A' \to B'$ were two morphisms, it could be the case that $Ff = Ff'$. The following notion of isomorphism does require the complete functor to have an inverse.

Definition 3.16. An **isomorphism of categories** is a functor F whose object and arrow functions are bijections. Equivalently, it is a functor F such that there exists an *inverse* functor G such that $F \circ G$ and $G \circ F$ are the identity functor.

Unfortunately, this notion of isomorphism of categories is too strict in some cases. Sometimes, the two compositions $F \circ G$ and $G \circ F$ are not exactly the identity functor, but isomorphic to it in some sense yet to be made precise. We develop weaker notions in the next section.

3.2.2 Natural transformations

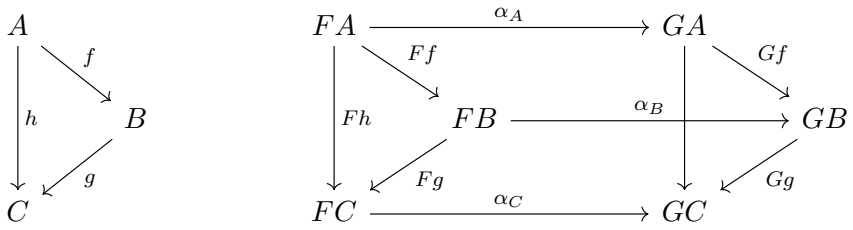
We have defined functors relating structures on different categories, and now, we could consider if any two given functors can be related in a *canonical* way, involving no arbitrary choices. An example is the relation between the identity functor and the double dual functor in vector spaces. They are related by the canonical isomorphism that sends each vector to the operator that evaluates a function over it. In this case, the isomorphism can be described without making any explicit reference to the space. On the other hand, the isomorphism relating a vector space to its dual depends on the choice of a basis. A family of *canonical* linear isomorphisms $\sigma_V\colon V \to V^*$ translating between an space and its dual, should be invariant to linear maps $\mu\colon V \to W$, so it should satisfy $\sigma_V(v)(w) = \sigma_W(\mu(v))(\mu(w))$, but this is not possible for all μ linear. The notion of *natural transformation* formalizes this intuitive idea.

A **natural transformation** between two functors F and G with the same domain and codomain, written as $\alpha\colon F \Rightarrow G$, is a family of morphisms parameterized by the objects of the domain category, $\{\alpha_C\colon FC \to GC\}_{C \in \mathcal{C}}$, such that $\alpha_{C'} \circ Ff = Gf \circ \alpha_C$ for every arrow $f\colon C \to C'$. That is, the following diagram commutes.

$$\begin{array}{ccc} C & & FC & \xrightarrow{\alpha_C} & GC \\ \downarrow{\scriptstyle f} & & {\scriptstyle Ff}\downarrow & & \downarrow{\scriptstyle Gf} \\ C' & & FC' & \xrightarrow{\alpha_{C'}} & GC' \end{array}$$

Sometimes, we also say that the family of morphisms α is *natural* in its argument. This

naturality property is what allows us to "translate" a commutative diagram from a functor to another, as in the following example.

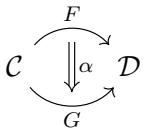


Natural isomorphisms are natural transformations in which every component, every morphism of the parameterized family, is invertible. In this case, the inverses form themselves a new transformation, whose naturality follows from the naturality of the original transformation. We say that F and G are *naturally isomorphic*, $F \cong G$, when there is a natural isomorphism between them.

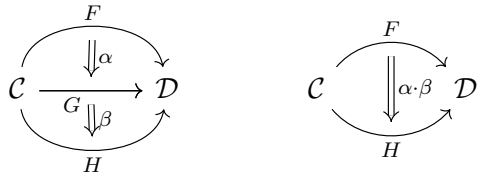
The notion of a natural isomorphism between functors allows us to weaken the condition of strict equality we imposed when talking about isomorphisms of categories (Definition 3.16). We say that two categories $\mathcal{C}$ and $\mathcal{D}$ are **equivalent** if there exist two functors $F\colon \mathcal{C} \to \mathcal{D}$ and $G\colon \mathcal{D} \to \mathcal{C}$ such that $G \circ F \cong \mathrm{Id}_{\mathcal{C}}$ and $F \circ G \cong \mathrm{Id}_{\mathcal{D}}$. The pair of functors endowed with the two natural isomorphisms is called an *equivalence of categories*

3.2.3 Composition of natural transformations

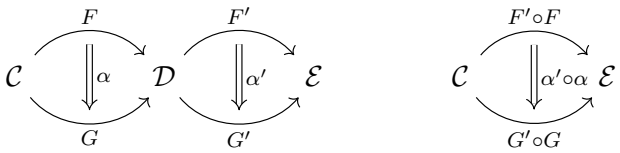
The next reasonable step is to ask how natural transformations compose. If we draw natural transformations between functors as double arrows, as in the following diagram,



we notice that two different notions of composition arise; we have a *vertical* composition of natural transformations, which, diagramatically, composes the two sequential natural transformations on the left side into a single transformation on the right side of the following diagram;

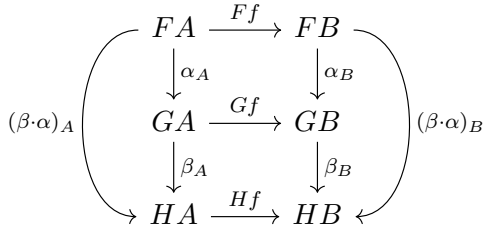


and we have a *horizontal* composition of natural transformations, which composes the two parallel natural transformations on the left side into a single transformation on the right side of the following diagram.

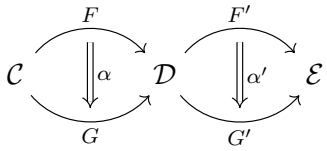


Definition 3.17. The **vertical composition** of two natural transformations $\alpha : F \Rightarrow G$ and $\beta : G \Rightarrow H$, denoted by $\beta \cdot \alpha$ is the family of morphisms defined by the objectwise composition of the morphisms of the two natural transformations. That is, $(\beta \cdot \alpha) = \{\beta_C \circ \alpha_C\}_{C \in \mathcal{C}}$.

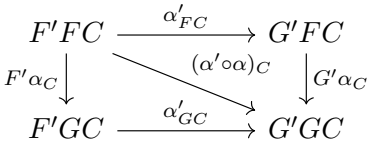
Naturality of this family follows from the naturality of its two factors. Given any morphism $f\colon A \to B$, the commutativity of the external square on the diagram below follows from the commutativity of the two internal squares.



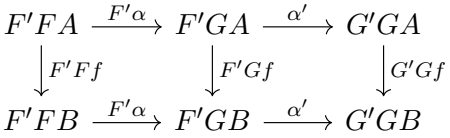
Definition 3.18 (Horizontal composition of natural transformations)**.** The **horizontal composition** of two natural transformations $\alpha\colon F \to G$ and $\alpha'\colon F' \to G'$, with domains and codomains as in the following diagram



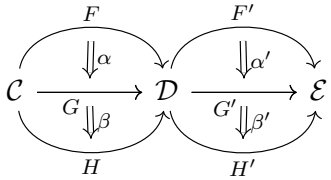
is denoted by $\alpha' \circ \alpha\colon F' \circ F \to G' \circ G$ and is defined as the family of morphisms given by $\alpha' \circ \alpha = \{G'\alpha_C \circ \alpha'_{FC}\}_{C \in \mathcal{C}} = \{\alpha'_{GC} \circ F'\alpha_C\}_{C \in \mathcal{C}}$, that is, by the diagonal of the following square, which is commutative by the naturality of α'.



Naturality of this family follows again from the naturality of its two factors. Given any morphism $f\colon A \to B$, the following diagram is commutative because it is the composition of two naturality squares given by the naturality of $F'\alpha$ and α'.



Proposition 3.19 (Interchange law)**.** *The two possible ways of composing vertical and horizontal transformations in a diagram like the following one*



are actually equivalent. That is, $(\beta' \cdot \alpha') \circ (\beta \cdot \alpha) = (\beta' \circ \beta) \cdot (\alpha' \circ \alpha)$.

Proof. By naturality of α', we have

$$\begin{aligned}(\beta' \cdot \alpha') \circ (\beta \cdot \alpha) &= \{\beta'_{HC} \circ \alpha'_{HC} \circ F'\beta_C \circ F'\alpha_C\}_{C \in \mathcal{C}} \\ &= \{\beta'_{HC} \circ G'\beta_C \circ \alpha'_{GC} \circ F'\alpha_C\}_{C \in \mathcal{C}} \\ &= (\beta' \circ \beta) \cdot (\alpha' \circ \alpha). \quad \square\end{aligned}$$

3.3 Constructions on categories

Before continuing our study of functors and universal properties, we provide some constructions and examples of categories we will need in the future. In particular, we consider some variations on the definition of functors and we use them to construct new categories.

3.3.1 Product categories

The **product category** (see [EM45]) of two categories $\mathcal{C}$ and $\mathcal{D}$, denoted by $\mathcal{C} \times \mathcal{D}$, is their product object in the category Cat. Explicitly, it is given by

- objects of the form $\langle C, D\rangle$, where $C \in \mathcal{C}$ and $D \in \mathcal{D}$;
- and morphisms of the form $\langle f, g\rangle : \langle C, D\rangle \to \langle C', D'\rangle$, 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 $\langle f', g'\rangle \circ \langle f, g\rangle = \langle f' \circ f, g' \circ g\rangle$. The definition of the product also provides *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 $Q\langle f, g\rangle = g$.

The **product functor** of two functors $F\colon \mathcal{C} \to \mathcal{C}'$ and $G\colon \mathcal{D} \to \mathcal{D}'$ is the unique functor $F \times G\colon \mathcal{C} \times \mathcal{D} \to \mathcal{C}' \times \mathcal{D}'$ making the following diagram commute.

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

Explicitly, it acts on arrows as $(F \times G)\langle f, g\rangle = \langle Ff, Gg\rangle$. In this sense, the $\times$ operation could be seen as a *functor with two arguments* acting on objects and morphisms of the Cat category. We now formalize this idea of a functor in two variables.

Bifunctors are functors from a product category, but they can also be regarded as functors on two variables. As we will show in the following proposition, they are completely determined by the two families of functors we obtain when we fix any of the arguments. We will also study a characterization of naturality between these functors.

Proposition 3.20 (Conditions for the existence of bifunctors)**.** *Let $\mathcal{B}, \mathcal{C}, \mathcal{D}$ categories with two families of functors*

$$\{F_C\colon \mathcal{B} \to \mathcal{D}\}_{C \in \mathcal{C}} \quad \text{and} \quad \{G_B\colon \mathcal{C} \to \mathcal{D}\}_{B \in \mathcal{B}},$$

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

$$G_{B'}g \circ F_C f = F_{C'}f \circ G_B g.$$

Proof. If the equality holds, the bifunctor can be defined as $S(b,c) = G_B(c) = F_C(b)$ in objects and as $S(f,g) = G_{B'}g \circ F_C f = F_{C'}f \circ G_B g$ on morphisms. This bifunctor preserves identities, as $S(\mathrm{id},\mathrm{id}) = G_B(\mathrm{id}) \circ F_C(\mathrm{id}) = \mathrm{id} \circ \mathrm{id} = \mathrm{id}$; and it preserves composition, as, for any morphisms f, f', g, g' with suitable domain and codomain, we have

$$S(f',g') \circ S(f,g) = Gg' \circ Ff' \circ Gg \circ Ff = Gg' \circ Gg \circ Ff' \circ Ff = S(f' \circ f, g' \circ g).$$

On the other hand, if a bifunctor exists, the condition is satisfied because

$$\begin{aligned} G_{B'}(g) \circ F_C(f) &= S(\mathrm{id}_{B'}, g) \circ S(f, \mathrm{id}_C) = S(\mathrm{id}_{B'} \circ f, g \circ \mathrm{id}_C) \\ &= S(f \circ \mathrm{id}_B, \mathrm{id}_{C'} \circ g) = S(f, \mathrm{id}_{C'}) \circ S(\mathrm{id}_B, g) \\ &= F_{C'}(f) \circ G_B(f). \quad \square \end{aligned}$$

Proposition 3.21 (Naturality for bifunctors)**.** *Naturality on both components of a bifunctor is equivalent to naturality in the usual sense. Given S, S' bifunctors, the family $\alpha_{B,C} \colon S(B,C) \to S'(B,C)$ is a natural transformation if and only if $\alpha_{B,C}$ is natural in B for each C and natural in C for each B.*

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

$$\begin{array}{ccc} S(B,C) & \xrightarrow{\alpha} & S'(B,C) \\ {\scriptstyle S\langle f,\mathrm{id}\rangle}\downarrow & & \downarrow{\scriptstyle S'\langle f,\mathrm{id}\rangle} \\ S(B',C) & \xrightarrow{\alpha} & S'(B',C) \end{array} \qquad \begin{array}{ccc} S(B,C) & \xrightarrow{\alpha} & S'(B,C) \\ {\scriptstyle S\langle \mathrm{id},g\rangle}\downarrow & & \downarrow{\scriptstyle S'\langle \mathrm{id},g\rangle} \\ S(B,C') & \xrightarrow{\alpha} & S'(B,C') \end{array}$$

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

$$\begin{array}{ccc} S(B,C) & \xrightarrow{\alpha} & S'(B,C) \\ {\scriptstyle S\langle f,\mathrm{id}\rangle}\downarrow & & \downarrow{\scriptstyle S'\langle f,\mathrm{id}\rangle} \\ S(B',C) & \xrightarrow{\alpha} & S'(B',C) \\ {\scriptstyle S\langle \mathrm{id},g\rangle}\downarrow & & \downarrow{\scriptstyle S'\langle \mathrm{id},g\rangle} \\ S(B',C') & \xrightarrow{\alpha} & S'(B',C') \end{array}$$

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

3.3.2 Opposite categories and contravariant functors

For many constructions in categories, it makes sense to consider how the process of *reverting all the arrows* yields a new construction.

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

Reversing all the arrows is a process that directly translates every property of the category into its *dual* property. A morphism f is a monomorphism if and only if f^{op} is an epimorphism;

a terminal object in $\mathcal{C}$ is an initial object in $\mathcal{C}^{op}$ and a right inverse becomes a left inverse on the opposite category. This process is also an *involution*, where $(f^{op})^{op}$ can be seen as f, and $(\mathcal{C}^{op})^{op}$ is trivially isomorphic to $\mathcal{C}$.

Definition 3.22 (Contravariant functor)**.** A **contravariant** functor from $\mathcal{C}$ to $\mathcal{D}$ is a functor from the opposite category, that is, $F\colon \mathcal{C}^{op} \to \mathcal{D}$. Non-contravariant functors are often called **covariant** functors, to emphasize the difference.

Example 3.23 (Hom functors)*.* In a locally small category $\mathcal{C}$, the **Hom-functor** is the bifunctor $\hom\colon \mathcal{C}^{op} \times \mathcal{C} \to \mathsf{Set}$, defined as $\hom(A, B)$ for any two objects $A, B \in \mathcal{C}$. Given $f\colon A \to A'$ and $g\colon B \to B'$, this functor is defined on any $p \in \hom(A, B)$ as postcomposition and precomposition, $\hom(f, g)(p) = f \circ p \circ g \in \hom(A', B')$. Partial applications of the functor give rise to

- $\hom(A, -)$, a covariant functor for any fixed $A \in \mathcal{C}$ that maps $g\colon B \to B'$ to the precomposition $- \circ g\colon \hom(A, B) \to \hom(A, B')$;
- $\hom(-, B)$, a contravariant functor for any fixed $B \in \mathcal{C}$ that maps $f\colon A \to A'$ to the postcomposition $f \circ -\colon \hom(A', B) \to \hom(A, B)$.

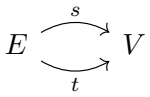
Note that this is a well-defined bifunctor by virtue of Proposition 3.20 and the fact that $(- \circ g) \circ (f \circ -) = (f \circ -) \circ (- \circ g)$ by associativity. This kind of functor, contravariant on the first variable and covariant on the second, is usually called a **profunctor**.

3.3.3 Functor categories

Given two categories $\mathcal{B}, \mathcal{C}$, the **functor category** $\mathcal{B}^{\mathcal{C}}$ has all functors from $\mathcal{C}$ to $\mathcal{B}$ as objects and natural transformations between them as morphisms. If we consider the category of small categories Cat, there is a hom-profunctor $-^{-}\colon \mathsf{Cat}^{op} \times \mathsf{Cat} \to \mathsf{Cat}$ sending any two categories $\mathcal{B}$ and $\mathcal{C}$ to their functor category $\mathcal{B}^{\mathcal{C}}$. Many mathematical constructions are examples of functor catgeories. In [LS09], multiple examples of usual mathematical constructions in terms of functor categories can be found. Graphs, for instance, can be seen as functors; and graphs homomorphisms as the natural transformations between them.

Example 3.24 (Graphs as functors)*.* We consider the category given by two objects and two non-identity morphisms,

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



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

Example 3.25 (Dynamical systems as functors). A set endowed with an endomorphism (S, α) can be regarded as a *dynamical system* in an informal way. Each state of the system is represented by an element of the set and and the transition function is represented by the endomorphism. That is, if we start at an initial state $s \in S$ and the transition function is given by $\alpha\colon S \to S$, the evolution of the system will be given by

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

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

This structure can be described as a functor from the monoid of natural numbers with addition, seen as a category. Note that any functor $D\colon \mathbb{N} \to \mathsf{Set}$ has to choose a set S, and an image for the morphism $(1+) : \mathbb{N} \to \mathbb{N}$, of the form $\alpha\colon S \to S$. The image of any natural number n is now determined by the image of $(1+)$ because the functor must preserve composition and identities; if $D(1+) = \alpha$, it follows that $D(t+) = \alpha^t$, where $\alpha^0 = \mathrm{id}$.

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

$$\begin{array}{ccc} S & \xrightarrow{f} & T \\ {\scriptstyle \alpha^n}\downarrow & & \downarrow{\scriptstyle \beta^n} \\ S & \xrightarrow{f} & T \end{array}$$

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

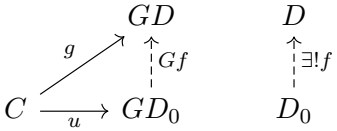
A further generalization is now possible: if we want to consider continuously-evolving dynamical systems, we can define functors from the monoid of real numbers under adition, that is, functors $\mathbb{R} \to \mathsf{Set}$. Note that these functors are given by a set S and a family of morphisms $\{\alpha_r\}_{r\in\mathbb{R}}$ such that $\alpha_r \circ \alpha_s = \alpha_{r+s}$ for all $r, s \in \mathbb{R}$. This example is discussed in [LS09].

3.4 Universality and limits

3.4.1 Universal arrows

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

A **universal arrow** from $C \in \mathcal{C}$ to $G\colon \mathcal{D} \to \mathcal{C}$ is a morphism $u\colon C \to GD_0$ such that for every $g\colon C \to GD$ exists a unique morphism $f\colon D_0 \to D$ making the following diagram commute.



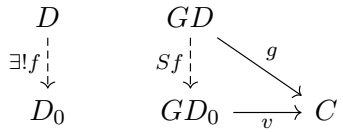
Note how an universal arrow is, equivalently, the initial object of the comma category $(C \downarrow G)$. Thus, universal arrows must be unique up to isomorphism.

Proposition 3.26 (Universality in terms of hom-sets)**.** *The arrow $u\colon C \to GD_0$ is universal if and only if $f \mapsto Gf \circ u$ is a bijection $\hom(D_0, D) \cong \hom(C, GD)$ natural in D. Any natural bijection of this kind is determined by a unique universal arrow.*

Proof. On the one hand, given an universal arrow, bijectivity follows from the definition of universal arrow; and naturality follows from the fact that $G(g \circ f) \circ u = Gg \circ Gf \circ u$. On the other hand, given a bijection φ, we define $u = \varphi(\mathrm{id}_{D_0})$. By naturality, we have the bijection $\varphi(f) = Gf \circ u$; and every arrow can be written in this form. □

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

A *dual* **universal arrow** from G to C is a morphism $v\colon GD_0 \to C$ such that for every $g\colon GD \to C$ exists a unique morphism $f\colon D \to D_0$ making the following diagram commute.



3.4.2 Representability

A **representation** of a functor from a locally small category to the category of sets, $K\colon \mathcal{D} \to \mathsf{Set}$, is a natural isomorphism $\psi\colon \hom_{\mathcal{D}}(r, -) \cong K$ making it isomorphic to a partially applied hom-functor. A functor is *representable* if it has a representation and the object r used in the representation is called its *representing object*.

Proposition 3.27 (Representations in terms of universal arrows)**.** *If $u\colon 1 \to Kr$ is a universal arrow for a functor $K\colon \mathcal{D} \to \mathsf{Set}$, then the map $f \mapsto Kf(u(*))$ is a representation. Every representation is obtained in this way.*

Proof. There is a trivial natural isomorphism $\hom_{\mathsf{Set}}(1, X) \Rightarrow X$ for each X; in particular $\hom_{\mathsf{Set}}(1, K-) \Rightarrow K-$ is a natural isomorphism. Every representation is then built as

$$\hom_{\mathcal{D}}(r, -) \cong \hom_{\mathsf{Set}}(1, K-) \cong K,$$

using universality of u. Moreover, every natural isomorphism $\hom_{\mathcal{D}}(r, -) \cong K$ determines an natural isomorphism $\hom_{\mathcal{D}}(r, -) \cong \hom_{\mathsf{Set}}(1, K-)$, which in turn determines a universal arrow by Proposition 3.26. □

3.4.3 Yoneda Lemma

The Yoneda lemma is one of the first results in pure category theory. It allows us to embed any small category into a functor category over the category of sets. It will be very useful when studying presheaves and algebraic theories (see Section 4.2.1) to create models of these theories.

Lemma 3.28 (Yoneda Lemma)**.** *For any $K\colon \mathcal{D} \to \mathsf{Set}$ and $r \in \mathcal{D}$, there is a bijection $y\colon \mathrm{Nat}(\hom_{\mathcal{D}}(r,-), K) \cong Kr$ sending any natural transformation $\alpha\colon \hom_{\mathcal{D}}(r,-) \Rightarrow K$ to its image on the identity, $\alpha_r(\mathrm{id}_r)$.*

Proof. The complete natural transformation α is determined by $\alpha_r(\mathrm{id}_r)$. By naturality, given any $f\colon r \to s$, it must be the case that $\alpha_s(f) = Kf(\alpha_r(\mathrm{id}_r))$.

$$\begin{array}{ccc} \hom(r,r) & \xrightarrow{\alpha_r} & Kr \\ & \mathrm{id}_r \longmapsto \alpha_r(\mathrm{id}_r) & \\ {\scriptstyle f\circ -}\downarrow & \downarrow \quad\quad \downarrow & \downarrow{\scriptstyle Kf} \\ & f \longmapsto \alpha_s(f) & \\ \hom(r,s) & \xrightarrow[\alpha_s]{} & Ks \end{array}$$

□

Corollary 3.29 (Characterization of natural transformations between representable functors)**.** *Given $r, s \in \mathcal{D}$, any natural transformation $\hom(r,-) \Rightarrow \hom(s,-)$ is of the form $- \circ h$ for a unique morphism $h\colon s \to r$.*

Proof. Using Yoneda Lemma (Lemma 3.28), we know that

$$\mathrm{Nat}(\hom_{\mathcal{D}}(r,-), \hom_{\mathcal{D}}(s,-)) \cong \hom_{\mathcal{D}}(s,r),$$

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

Proposition 3.30 (Naturality of the Yoneda Lemma)**.** *The bijection on the Yoneda Lemma (Lemma 3.28), $y\colon \mathrm{Nat}(\hom_{\mathcal{D}}(r,-), K) \cong Kr$, is a natural isomorphism between two functors $\mathsf{Set}^{\mathcal{D}} \times \mathcal{D} \to \mathsf{Set}$.*

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

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

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

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

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

$$\begin{array}{ccc} \mathrm{Nat}(\hom(r,-), K) & \xrightarrow{y} & Kr \\ {\scriptstyle N\langle f,F\rangle}\downarrow & & \downarrow{\scriptstyle E\langle f,F\rangle} \\ \mathrm{Nat}(\hom(r',-), K') & \xrightarrow{y} & K'r' \end{array}$$

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

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

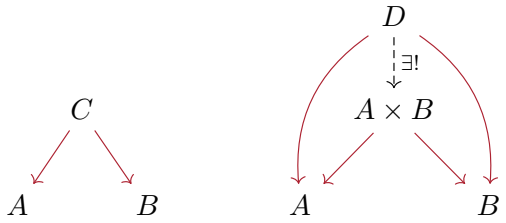
Definition 3.31. In the conditions of the Yoneda Lemma (Lemma 3.28) the **Yoneda functor**, $Y\colon \mathcal{D}^{op} \to \mathsf{Set}^{\mathcal{D}}$, is defined with the arrow function

$$(f\colon s \to r) \mapsto \Big(\hom_{\mathcal{D}}(f,-)\colon \hom_{\mathcal{D}}(r,-) \to \hom_{\mathcal{D}}(s,-)\Big).$$

It can be also written as $Y\colon \mathcal{D} \to \mathsf{Set}^{\mathcal{D}^{op}}$. By the Yoneda Lemma, there is a bijection $y\colon \mathrm{Nat}(\hom(r,-),\hom(s,-)) \cong \hom(s,r)$ given by $y(\hom(f,-)) = f$; making the Yoneda functor full and faithful.

3.4.4 Limits

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



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

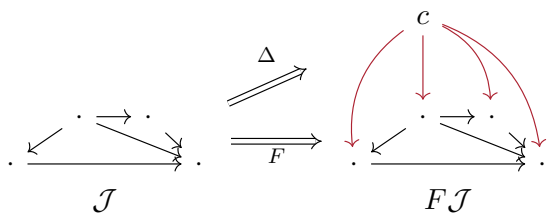
We will be able to create new constructions on categories by formalizing the notion of cone and generalizing to arbitrary bases, given as functors from arbitrary index categories. Constant functors are the first step into formalizing the notion of *cone*.

Definition 3.32 (Constant functor)**.** The **constant functor** $\Delta\colon \mathcal{C} \to \mathcal{C}^{\mathcal{J}}$ sends each object $c \in \mathcal{C}$ to a constant functor $\Delta c\colon \mathcal{J} \to \mathcal{C}$ defined as

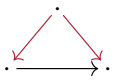
- the constantly-c function for objects, $\Delta(j) = c$;
- and the constantly-id_c function for morphisms, $\Delta(f) = \mathrm{id}_c$.

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

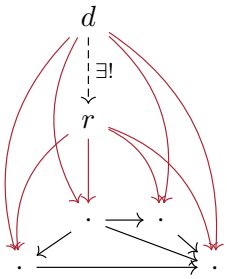
We could say that Δc compresses the whole category $\mathcal{J}$ into c. A natural transformation from this functor to some other $F\colon \mathcal{J} \to \mathcal{C}$ should be regarded as a ***cone*** from the object c to a copy of $\mathcal{J}$ inside the category $\mathcal{C}$; as the following diagram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



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



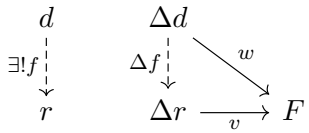
on that cone must be commutative. Thus, natural transformations are a way to recover all the information of an arbitrary ***index category*** $\mathcal{J}$ that was encoded in c by the constant functor. As we did with products, we want to find the cone that best encodes that information; a universal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



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

Definition 3.33 (Limit)**.** The **limit** of a functor $F\colon \mathcal{J} \to \mathcal{C}$ is an object $r \in \mathcal{C}$ such that there exists a universal arrow $v\colon \Delta r \Rightarrow F$ from Δ to F. It is usually written as $r = \varprojlim F$.

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



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

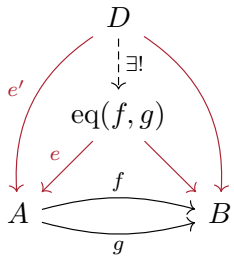
By choosing different index categories, we will be able to define multiple different constructions on categories as limits.

3.4.5 Examples of limits

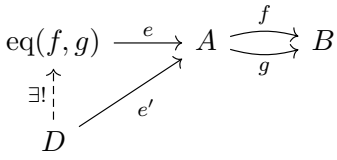
For our first example, we will take the following category, called $\downarrow\downarrow$ as index category,

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

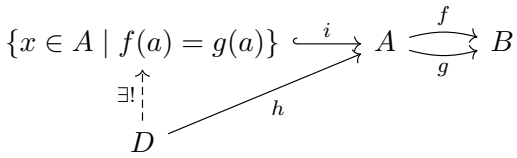
any other object with a similar morphism factorizes uniquely through it



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

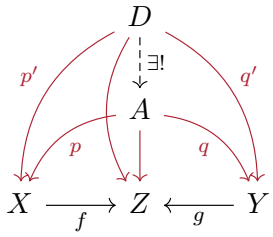


Example 3.34 (Equalizers in Sets). The equalizer of two parallel functions $f, g \colon A \to B$ in Set is $\{x \in A \mid f(a) = g(a)\}$ with the inclusion morphism. Given any other function $h \colon D \to A$ such that $f \circ h = g \circ h$, we know that $f(h(d)) = g(h(d))$ for any $d \in D$. Thus, h can be factorized through the equalizer.



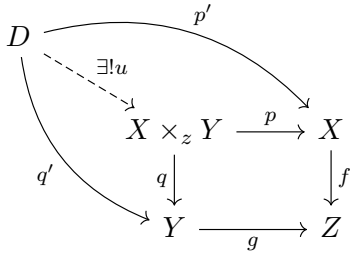
Example 3.35 (Kernels). In the category of abelian groups, the kernel of a function f, $\ker(f)$, is the equalizer of $f \colon G \to H$ and a function sending each element to the zero element of H. The same notion of kernel can be defined in the category of R-Modules, for any ring R.

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



Again, the central arrow of the diagram is determined as $f \circ q = g \circ p$; so it can be ommited in the diagram. The usual definition of a pullback for two morphisms $f \colon X \to Z$ and $g \colon Y \to Z$ is the universal pair of morphisms $p \colon A \to X$ and $q \colon A \to Y$ such that $f \circ q = g \circ p$, that is, given any pair of morphisms $p' \colon D \to X$ and $q' \colon D \to Y$, there exists a unique $u \colon D \to A$ making the diagram commute. Usually we write the pullback object as $X \times_Z Y$ and we write

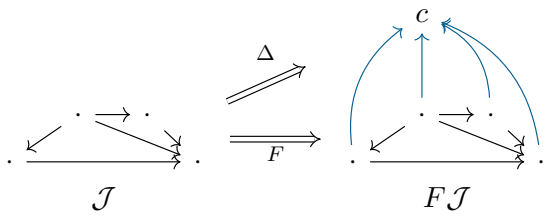
this property diagramatically as



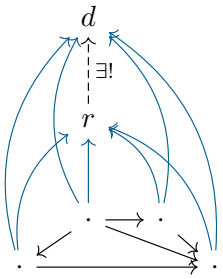
The square in this diagram is usually called a *pullback square*, and the pullback object is usually called a *fibered product*.

3.4.6 Colimits

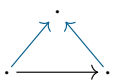
A colimit is the dual notion of a limit. We could consider cocones to be the dual of cones and pick the universal one. Once an index category $\mathcal{J}$ and a base category $\mathcal{C}$ are fixed, a ***cocone*** is a natural transformation from a functor on the base category to a constant functor.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



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



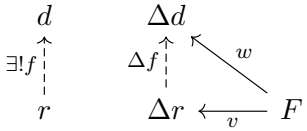
and naturality implies that each triangle



commutes.

Definition 3.36 (Colimits)**.** The **colimit** of a functor $F\colon J \to \mathcal{C}$ is an object $r \in \mathcal{C}$ such that there exists a universal arrow $u\colon F \Rightarrow \Delta r$ from F to Δ. It is usually written as $r = \varinjlim F$.

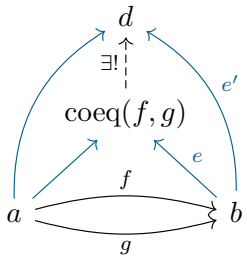
That is, for every natural transformation $w\colon F \Rightarrow \Delta d$, there is a unique morphism $f\colon r \to d$ such that



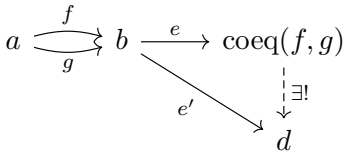
commutes. This reflects directly on the universality of the cocone we described earlier and proves that colimits are unique up to isomorphism.

3.4.7 Examples of colimits

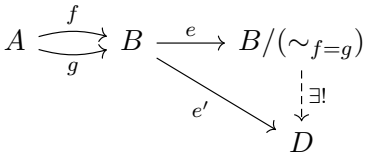
Coequalizers are the dual of *equalizers*; colimits of functors from $\downarrow\downarrow$. The coequalizer of two parallel arrows is an object $\operatorname{coeq}(f, g)$ with a morphism $e\colon b \to \operatorname{coeq}(f, g)$ such that $e \circ f = e \circ g$; and such that any other object with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



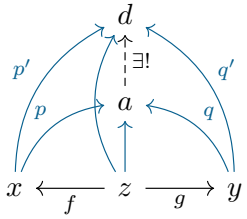
as the right part of the cocone is completely determined by the left one, the diagram can be written as



Example 3.37 (Coequalizers in Sets). The coequalizer of two parallel functions $f, g\colon A \to B$ in Set is $B/(\sim_{f=g})$, where $\sim_{f=g}$ is the minimal equivalence relation in which we have $f(a) \sim g(a)$ for each $a \in A$. Given any other function $h\colon B \to D$ such that $h(f(a)) = h(g(a))$, it can be factorized in a unique way by $h'\colon B/\sim_{f=g} \to D$.

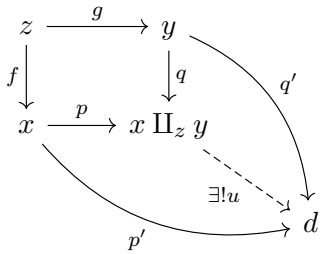


Pushouts are the dual of pullbacks; colimits whose index category is $\cdot \leftarrow \cdot \rightarrow \cdot$, that is, the dual of the index category for pullbacks. Diagramatically,



and we can define the pushout of two morphisms $f\colon z \to x$ and $g\colon z \to y$ as a pair of morphisms $p\colon x \to a$ and $q\colon y \to a$ such that $p \circ f = q \circ g$ which are also universal, that is, given any pair

of morphisms $p' \colon x \to d$ and $q' \colon y \to d$, there exists a unique $u \colon a \to d$ making the diagram commute.



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

3.5 Adjoints, monads and algebras

3.5.1 Adjunctions

An **adjunction** from categories $\mathcal{X}$ to $\mathcal{Y}$ is a pair of functors $F \colon \mathcal{X} \to \mathcal{Y}$, $G \colon \mathcal{Y} \to \mathcal{X}$ with a bijection $\varphi \colon \hom(FX, Y) \cong \hom(X, GY)$, natural in both $X \in \mathcal{X}$ and $Y \in \mathcal{Y}$. We say that F is *left-adjoint* to G and that G is *right-adjoint* to F and we denote this as $F \dashv G$.

Naturality of φ, by Proposition 3.21, means that both

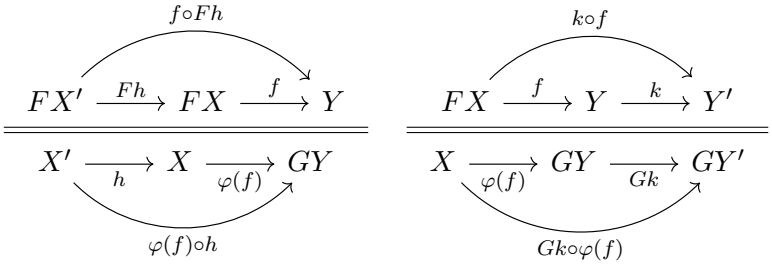
$$\begin{array}{ccc} \hom(FX, Y) & \xrightarrow{\varphi} & \hom(X, GY) \\ {\scriptstyle -\circ Fh}\downarrow & & \downarrow{\scriptstyle -\circ h} \\ \hom(FX', Y) & \xrightarrow[\varphi]{} & \hom(X', GY) \end{array} \qquad \begin{array}{ccc} \hom(FX, Y) & \xrightarrow{\varphi} & \hom(X, GY) \\ {\scriptstyle k\circ -}\downarrow & & \downarrow{\scriptstyle Gk\circ -} \\ \hom(FX, Y') & \xrightarrow[\varphi]{} & \hom(X, GY') \end{array}$$

commute for every $h \colon X \to X'$ and $k \colon Y \to Y'$. That is, for every $f \colon FX \to Y$, we have $\varphi(f) \circ h = \varphi(f \circ Fh)$ and $Gk \circ \varphi(f) = \varphi(k \circ f)$. Equivalently, φ^{-1} is natural, and that means that, for every $g \colon X \to GY$, we have $k \circ \varphi^{-1}(g) = \varphi^{-1}(Gk \circ g)$ and $\varphi^{-1}(g) \circ Fh = \varphi^{-1}(g \circ h)$.

A different and maybe more intuitive way to write adjunctions is by logical diagrams (see [Lawb]). An adjunction $F \dashv G$ can be written as

$$\frac{FX \xrightarrow{f} Y}{X \xrightarrow{\varphi(f)} GY}$$

to emphasize that, for each morphism $f \colon FX \to Y$, there exists a unique morphism $\varphi(f) \colon X \to GY$; written in a way that resembles bidirectional logical inference rules. Naturality, in this setting, means that precomposition and postcomposition of arrows are preserved by the *inference rule*. Given morphisms $h \colon X' \to X$ and $k \colon Y \to Y'$, we know by naturality that the composed arrows of the following diagrams are adjoint to one another.



In other words, $\varphi(f) \circ h = \varphi(f \circ Fh)$ and $Gk \circ \varphi(f) = \varphi(k \circ f)$, as we wrote earlier. In the following two propositions, we will characterize all this information in terms of natural transformations made up of universal arrows. Given an adjunction $F \dashv G$, we define

- the **unit** η as the family of morphisms $\eta_X = \varphi(\mathrm{id}_{FX})\colon X \to GFX$, for each X;
- the **counit** ε as the family of morphisms $\varepsilon_Y = \varphi^{-1}(\mathrm{id}_{GY})\colon FGY \to Y$, for each Y.

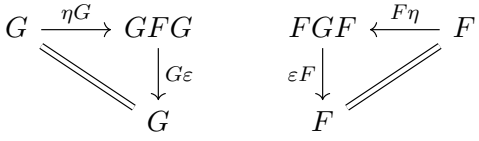
Diagramatically, they can be obtained by taking Y to be FX and X to be GY, respectively, in the definition of adjunction.

$$\frac{FX \xrightarrow{\mathrm{id}} FX}{X \xrightarrow{\eta_x} GFX} \qquad \frac{FGY \xrightarrow{\varepsilon_y} Y}{GY \xrightarrow{\mathrm{id}} GY}$$

Proposition 3.38 (Units and counits are natural transformations)**.** *The **unit** and the **counit** are natural transformations such that*

1. *for each $f\colon FX \to Y$, $\varphi(f) = Gf \circ \eta_x$;*
2. *for each $g\colon X \to GY$, $\varphi^{-1}(g) = \varepsilon_y \circ Fg$.*

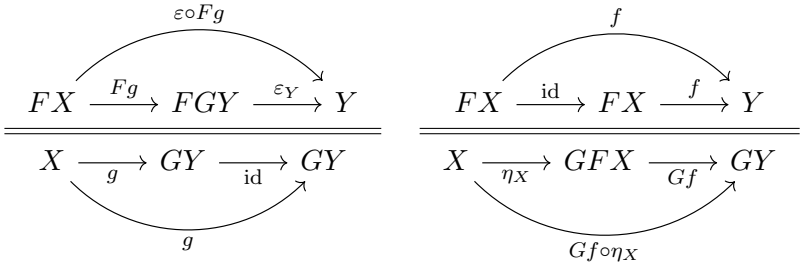
*Moreover, they follow the **triangle identities**, $G\varepsilon \circ \eta G = \mathrm{id}_G$ and $\varepsilon F \circ F\eta = \mathrm{id}_F$.*



Proof. The right and left adjunct formulas are particular instances of the naturality equations we gave in the definition of φ;

- $Gf \circ \eta = Gf \circ \varphi(\mathrm{id}) = \varphi(f \circ \mathrm{id}) = \varphi(f)$;
- $\varepsilon_y \circ Fg = \varphi^{-1}(\mathrm{id}) \circ Fg = \varphi^{-1}(\mathrm{id} \circ g) = \varphi^{-1}(g)$;

diagramatically,



The naturality of η and ε can be deduced again from the naturality of φ; given any two functions $h\colon X \to X'$ and $k\colon Y \to Y'$,

- $GFh \circ \eta_X = GFh \circ \varphi(\mathrm{id}_{FX}) = \varphi(Fh) = \varphi(\mathrm{id}_{FX'}) \circ h = \eta_{X'} \circ h$;
- $\varepsilon_{Y'} \circ FGk = \varphi^{-1}(\mathrm{id}_{GY'}) \circ FGk = \varphi^{-1}(Gk) = k \circ \varphi^{-1}(\mathrm{id}_{GY}) = k \circ \varepsilon_Y$;

diagramatically, we can prove that the adjunct of Fh is $GFh \circ \eta_X$ and $\eta'_X \circ h$ at the same time; while the adjunct of Gk is $k \circ \varepsilon_Y$ and $\varepsilon_{Y'} \circ FGk$ at the same time.

$$\begin{array}{c} X \xrightarrow{h} Y \xrightarrow{\eta} GFY \\ \hline\hline FX \xrightarrow{\mathrm{id}} FX \xrightarrow{Fh} FY \xrightarrow{\mathrm{id}} FY \\ \hline\hline X \xrightarrow{\eta} GFX \xrightarrow{GFh} GFY \end{array} \qquad \begin{array}{c} FGX \xrightarrow{\varepsilon} X \xrightarrow{k} Y \\ \hline\hline GX \xrightarrow{\mathrm{id}} GX \xrightarrow{Gk} GY \xrightarrow{\mathrm{id}} GY \\ \hline\hline FGx \xrightarrow{FGk} FGy \xrightarrow{\varepsilon} y \end{array}$$

Finally, the triangle identities follow directly from the previous ones: we have $\mathrm{id} = \varphi(\varepsilon) = G\varepsilon \circ \eta$ and $\mathrm{id} = \varphi^{-1}(\eta) = \varepsilon \circ F\eta$. □

Proposition 3.39 (Characterization of adjunctions)**.** *Each adjunction is $F \dashv G$ between categories $\mathcal{X}$ and $\mathcal{Y}$ is completely determined by any of the following data,*

1. *functors F, G and $\eta\colon 1 \Rightarrow GF$ where $\eta_X\colon X \to GFX$ is universal to G.*
2. *functor G and universals $\eta_X\colon X \to GF_0X$, where $F_0X \in \mathcal{Y}$, creating a functor F.*
3. *functors F, G and $\varepsilon\colon FG \Rightarrow 1$ where $\varepsilon_Y\colon FGY \to Y$ is universal from F.*
4. *functor F and universals $\varepsilon_Y\colon FG_0Y \to Y$, where $G_0Y \in \mathcal{X}$, creating a functor G.*
5. *functors F, G, with natural transformations satisfiying the triangle identities $G\varepsilon \circ \eta G = \mathrm{id}$ and $\varepsilon F \circ F\eta = \mathrm{id}$.*

Proof. 1. Universality of η_X gives a isomorphism $\varphi\colon \hom(FX, Y) \cong \hom(X, GY)$ between the arrows in the following diagram

$$\begin{array}{ccc} & GY & \qquad Y \\ {\scriptstyle f}\nearrow & \uparrow{\scriptstyle Gg} & \qquad \uparrow{\scriptstyle \exists! g} \\ X \xrightarrow[\eta_x]{} & GFX & \qquad FX \end{array}$$

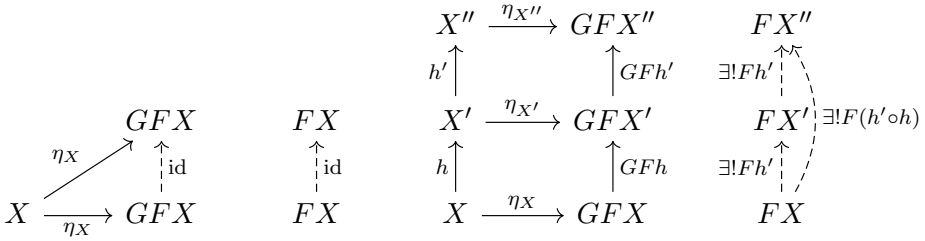
defined as $\varphi(g) = Gg \circ \eta_X$. This isomorphism is natural in X; for every $h\colon X' \to X$ we know by naturality of η that $Gg \circ \eta \circ h = G(g \circ Fh) \circ \eta$. The isomorphism is also natural in Y; for every $k\colon Y \to Y'$ we know by functoriality of G that $Gh \circ Gg \circ \eta = G(h \circ g) \circ \eta$.

2. We can define a functor F on objects as $FX = F_0X$. Given any $h\colon X \to X'$, we can use the universality of η to define Fh as the unique arrow making this diagram commute

$$\begin{array}{ccc} & GFX' & \qquad FX' \\ {\scriptstyle \eta_{X'} \circ h}\nearrow & \uparrow{\scriptstyle GFh} & \qquad \uparrow{\scriptstyle \exists! Fh} \\ X \xrightarrow[\eta_X]{} & GFX & \qquad FX \end{array}$$

and this choice makes F a functor and η a natural transformation, as it can be checked in the following diagrams using the existence and uniqueness given by the universality of η in both

cases.



3. The proof is dual to that of *1.*

4. The proof is dual to that of *2.*

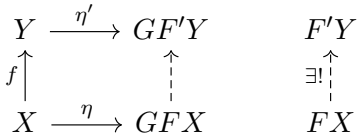
5. We can define two functions $\varphi(f) = Gf \circ \eta_X$ and $\theta(g) = \varepsilon_Y \circ Fg$. We checked in 1 (and 3) that these functions are natural in both arguments; now we will see that they are inverses of each other using naturality and the triangle identities

- $\varphi(\theta(g)) = G\varepsilon \circ GFg \circ \eta = G\varepsilon \circ \eta \circ g = g$;
- $\theta(\varphi(f)) = \varepsilon \circ FGf \circ F\eta = f \circ \varepsilon \circ F\eta = f$. □

Proposition 3.40 (Essential uniqueness of adjoints)**.** *Two adjoints to the same functor $F, F' \dashv G$ are naturally isomorphic.*

Proof. Note that the two different adjunctions give two units η, η', and for each X both $\eta_X \colon X \to GFX$ and $\eta_{X'} \colon X \to GF'X$ are universal arrows from X to G. As universal arrows are unique up to isomorphism, we have a unique isomorphism $\theta_X \colon FX \to F'X$ such that $G\theta_X \circ \eta_X = \eta'_X$.

We know that θ is natural because there are two arrows, $\theta \circ Ff$ and $F'f \circ \theta$, making this universal diagram commute



because

- $G(\theta \circ Ff) \circ \eta = G\theta \circ GFf \circ \eta = G\theta \circ \eta \circ f = \eta' \circ f$;
- $G(F'f \circ \theta) \circ \eta = GF'f \circ G\theta \circ \eta = GF'f \circ \eta' = \eta' \circ f$;

thus, they must be equal, $\theta \circ Ff = F'f \circ \theta$. □

Theorem 3.41 (Composition of adjunctions)**.** *Given two adjunctions $\varphi \colon F \dashv G$ and $\theta \colon F' \dashv G'$ between categories $\mathcal{X}, \mathcal{Y}$ and $\mathcal{Y}, \mathcal{Z}$ respectively, the composite functors yield a composite adjunction $\varphi \cdot \theta \colon F' \circ F \dashv G \circ G'$. Let the unit and counit of φ be $\langle \eta, \varepsilon \rangle$ and the unit and counit of θ be $\langle \eta', \varepsilon' \rangle$; the unit and counit of the composite adjunction are $\langle G\eta' F \circ \eta,\ \varepsilon' \circ F'\varepsilon G' \rangle$.*

Proof. We see that the vertical composition of two natural isomorphisms is itself an natural isomorphism because the composition of isomorphisms is itself an isomorphism. We compose as in the following diagram.

$$\frac{\dfrac{F'FX \xrightarrow{f} Y}{FX \xrightarrow{\theta(f)} G'Y}}{X \xrightarrow{\varphi\theta(f)} GG'Y}$$

If we apply the two natural isomorphisms to the identity, we find the unit and the counit of the adjunction.

$$\frac{\dfrac{F'FX \xrightarrow{\mathrm{id}} F'FX}{FX \xrightarrow{\mathrm{id}} FX \xrightarrow{\eta'_{FX}} G'F'FX}}{X \xrightarrow[\eta]{} GFX \xrightarrow[G\eta'_{FX}]{} GG'F'FX} \qquad \frac{\dfrac{GG'Z \xrightarrow{\mathrm{id}} GG'Z}{FGG'Z \xrightarrow{\varepsilon_{G'Z}} G'Z \xrightarrow{\mathrm{id}} G'Z}}{F'FGG'Z \xrightarrow[F'\varepsilon_{G'Z}]{} F'G'Z \xrightarrow[\varepsilon']{} Z}$$

□

3.5.2 Examples of adjoints

Example 3.42 (Product and coproduct as adjoints). Given any category $\mathcal{C}$, we define a **diagonal functor** to a product category $\Delta\colon \mathcal{C} \to \mathcal{C} \times \mathcal{C}$, sending every object X to a pair (X, X), and each morphism $f : X \to Y$ to the pair $\langle f, f\rangle\colon (X, X) \to (Y, Y)$.

The right adjoint to this functor will be the categorical product $\times\colon \mathcal{C} \times \mathcal{C} \to \mathcal{C}$, sending each pair of objects to their product and each pair of morphisms to their unique product. The left adjoint to this functor will be the categorical sum, $+\colon \mathcal{C} \times \mathcal{C} \to \mathcal{C}$, sending each pair of objects to their sum and each pair of morphisms to their unique sum. That is, we have the following chain of adjoints,

$$+ \dashv \Delta \dashv \times.$$

More precisely, knowing that a morphism $(X, X') \to (Y, Z)$ is actually pair of morphisms $X \to Y$ and $X' \to Z$, the adjoint properties for the diagonal functor

$$\frac{\Delta X \longrightarrow (Y, Z)}{X \longrightarrow \times(Y, Z)} \qquad \frac{+(X, Y) \longrightarrow Z}{(X, Y) \longrightarrow \Delta Z}$$

can be rewritten as bidirectional inference rules with two premises

$$\frac{X \to Y \qquad X \to Z}{X \longrightarrow Y \times Z} \qquad \frac{X + Y \longrightarrow Z}{X \to Z \qquad Y \to Z}$$

which are exactly the universal properties of the product and the sum. The necessary natural isomorphism is given by the existence and uniqueness provided by the inference rule.

Example 3.43 (Free and forgetful functors). Let Mon be the category of monoids with monoid homomorphisms. A functor $U\colon \mathsf{Mon} \to \mathsf{Set}$ can be defined by sending each morphism to its underlying set and each monoid homomorphism to its underlying function between sets. Funtors of this kind are called **forgetful functors**, as they simply *forget* part of the algebraic structure.

Left adjoints to forgetful functors are called **free functors**. In this case, the functor $F\colon \mathsf{Set} \to \mathsf{Mon}$ taking each set to the free monoid over it and each function to its unique extension to the free monoid. Note how it preserves identities and composition. The adjunction can be seen diagramatically as

$$\frac{FA \xrightarrow{\overline{f}} M}{A \xrightarrow{f} UM}$$

where each monoid homormorphism from the free monoid, $FA \to M$ can be seen as the unique extension of a function from the set of generators $f\colon A \to UM$ to a full monoid homomorphism, $\overline{f}\colon FA \to M$.

Note how, while this characterizes the notion of free monoid, it does not provide an explicit construction. Indeed, given $f\colon A \to UM$, we can take the free monoid FA to consist on the words over the elements of A endowed with the concatenation operator; the only way to extend f to an homomorphism is to define

$$\overline{f}(a_1 a_2 \dots a_n) = f(a_1) f(a_2) \dots f(a_n).$$

Note also that every homomorphism from the free monoid is determined by how it acts on the generator set. This notion of forgetful and free functors can be generalized to algebraic structures other than monoids.

3.5.3 Monads

The notion of *monads* is pervasive in category theory. A monad is a certain type of endofunctor that arises naturally when considering adjoints. They will be useful to model algebraic notions inside category theory and to model a variety of effects and contextual computations in functional programming languages. We will only prove here that each adjunction gives rise to a monad, but it is also true that there are multiple ways to turn a monad into an adjunction. With this result, we aim to illustrate how adjoints capture another fundamental concept in functional programming.

A **monad** is a functor $T\colon X \to X$ with natural transformations

- $\eta\colon \mathrm{Id} \Rightarrow T$, called *unit*; and
- $\mu\colon T^2 \Rightarrow T$, called *multiplication*;

such that the following two diagrams commute.

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

The first diagram is encoding some form of associativity of the multiplication, while the second one is encoding the fact that η creates a neutral element with respect to this multiplication. These statements will be made precise when we talk about algebraic theories. A **comonad** is the dual of a monad.

Proposition 3.44. *Given an adjunction $F \dashv G$, the composition 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

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

where the first diagram is commutative by Proposition 3.19 and the second one is obtained by applying functors G and F. Unit laws follow from the after applying F and G. □

3.5.4 Algebras

In category theory, the word *algebra* denotes certain structures that can be easily described in terms of endofunctors as universal fixed points for some functors. They model inductive constructions in terms of category theory.

Let $F\colon \mathcal{C} \to \mathcal{C}$ be a functor. An **F-algebra** is an object X with a map $\mu\colon FX \to X$ called *structure map*. A morphism between two F-algebras $\mu\colon FX \to X$ and $\nu\colon FY \to Y$ is a map $h : X \to Y$ such that the following diagram commutes.

$$\begin{array}{ccc} FX & \xrightarrow{Fh} & FY \\ \downarrow{\scriptstyle \mu} & & \downarrow{\scriptstyle \nu} \\ X & \xrightarrow{h} & Y \end{array}$$

This defines a category $\mathsf{Alg}_F(\mathcal{C})$. The initial object of this category, is called the **initial algebra** for F; it needs not to exist, but when it does, it is unique up to isomorphism.

F-algebras are closely related to induction principles. The following theorem states that the initial algebra is a fixed point of the functor, and, by its initiality property, it maps into any other fixed point.

Theorem 3.45 (Lambek's theorem)**.** *The structure map of an initial algebra is an isomorphism. That is, if X is an initial algebra, $\mu\colon FX \cong X$ (see [Awo10]).*

Proof. Consider the following commutative diagram, where $l\colon X \to FX$ is given by initiality of X.

$$\begin{array}{ccccc} FX & \xrightarrow{Fl} & FFX & \xrightarrow{F\mu} & FX \\ {\scriptstyle \mu}\downarrow & & \downarrow{\scriptstyle F\mu} & & \downarrow{\scriptstyle \mu} \\ X & \xrightarrow{l} & FX & \xrightarrow{\mu} & X \end{array}$$

By initiality of X, we have $\mu \circ l = \mathrm{id}$, and by commutativity of the left square, $l \circ \mu = F(\mu \circ l) = \mathrm{id}$. □

Example 3.46 (Natural numbers object). Consider the functor $F(X) = 1 + X$ in a category $\mathcal{C}$ with coproducts and a terminal object. Its initial algebra is called a **natural numbers object** due to the fact that, in Set, this initial algebra is precisely the set of natural numbers $\mathbb{N}$ with the successor function $\mathrm{succ}\colon \mathbb{N} \to \mathbb{N}$ and the zero element given as a morphism from

the terminal object, $0\colon 1 \to \mathbb{N}$.

$$\begin{array}{ccc} 1+\mathbb{N} & \longrightarrow & 1+X \\ {\scriptstyle\langle 0,\mathrm{succ}\rangle}\downarrow & & \downarrow{\scriptstyle\langle x,f\rangle} \\ \mathbb{N} & \xrightarrow{\varphi} & X \end{array}$$

Let X be an F-algebra given by $x\colon 1 \to X$ and $f\colon X \to X$; by induction over the natural numbers we can show that a morphism of algebras φ making that diagram commute must follow $\varphi(0) = x$ and $\varphi(\mathrm{succ}(n)) = f(\varphi(n))$. Thus, in a certain sense, initiality captures the principle of induction.

For instance, we can define addition $+\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, interpreted as a unary operation $+\colon \mathbb{N} \to \hom(\mathbb{N},\mathbb{N})$, as the unique morphism φ from the initial algebra to the algebra given by $\hom(\mathbb{N},\mathbb{N})$ with id and postcomposition with succ.

$$\begin{array}{ccc} 1+\mathbb{N} & \longrightarrow & 1+\hom(\mathbb{N},\mathbb{N}) \\ {\scriptstyle\langle 0,\mathrm{succ}\rangle}\downarrow & & \downarrow{\scriptstyle\langle \mathrm{id},\mathrm{succ}\circ -\rangle} \\ \mathbb{N} & \xrightarrow{+} & \hom(\mathbb{N},\mathbb{N}) \end{array}$$

This definition immediately implies the equalities $0 + m = \mathrm{id}(m) = m$ and $\mathrm{succ}(n) + m = (\mathrm{succ} \circ (n + \_))(m) = \mathrm{succ}(n + m)$.

Example 3.47 (Free monoids). Fixing a set A and changing the functor slightly to $F(X) = 1 + A \times X$ we get the set of lists of elements of A as the initial algebra. This algebra is written as $\mathrm{List}(A)$ with the empty list $\mathrm{nil}\colon 1 \to \mathrm{List}(A)$ and appending, $\mathrm{cons}\colon A \times \mathrm{List}(A) \to \mathrm{List}(A)$, as the binary operation.

$$\begin{array}{ccc} 1+A\times\mathrm{List}(A) & \longrightarrow & 1+A\times X \\ {\scriptstyle\langle \mathrm{nil},\mathrm{cons}\rangle}\downarrow & & \downarrow{\scriptstyle\langle x,\bullet\rangle} \\ \mathrm{List}(A) & \xrightarrow{\varphi} & X \end{array}$$

Given any other F-algebra X, with $x\colon 1 \to X$ and $\bullet\colon A \times X \to X$, we can show by induction on the length of the list that the unique morphism $\varphi\colon \mathrm{List}(A) \to X$ making the diagram commute must be such that $\varphi(\mathrm{nil}) = x$ and $\varphi(\mathrm{cons}(a, l)) = a \bullet \varphi(l)$.

For instance, fixing $A = \mathbb{N}$, we can define the sum of a list of naturals $\mathrm{sum}\colon \mathrm{List}(\mathbb{N}) \to \mathbb{N}$ as the unique morphism φ from the initial algebra to the algebra given by $\mathbb{N}$ with zero and the addition.

$$\begin{array}{ccc} 1+\mathbb{N}\times\mathrm{List}(\mathbb{N}) & \longrightarrow & 1+\mathbb{N}\times\mathbb{N} \\ {\scriptstyle\langle \mathrm{nil},\mathrm{cons}\rangle}\downarrow & & \downarrow{\scriptstyle\langle 0,+\rangle} \\ \mathrm{List}(\mathbb{N}) & \xrightarrow{\mathrm{sum}} & \mathbb{N} \end{array}$$

This definition immediately implies the equalities $\mathrm{sum}(\mathrm{nil}) = 0$ and $\mathrm{sum}(\mathrm{cons}(m, l)) = m + \mathrm{sum}(l)$.

Chapter 4

Categorical logic

4.1 Presheaves

Presheaves can be thought as sets parametrized by a category or as *generalized sets*. As we will study in the following chapters, categories of presheaves share a lot of categorical structure with the category of Sets and can be used as non-standard models of lambda calculus.

Specifically, a *presheaf* on the category $\mathcal{C}$ is a functor $S\colon \mathcal{C}^{op} \to \mathsf{Set}$ from the opposite category to the category of sets. The category of presheaves on $\mathcal{C}$ is a functor category of the form $\mathsf{Set}^{\mathcal{C}^{op}}$. In particular, the examples we gave when talking about functor catgeories (Examples 3.24 and 3.25) are presheaf categories.

4.2 Cartesian closed categories and lambda calculus

4.2.1 Lawvere theories

The usual notion of *algebraic theory* is given by a set of *k-ary operations* for each $k \in \mathbb{N}$ and certain axioms between the terms that can be constructed inductively using free veriables and these operations. For instance, the theory of groups is given by a binary operation $(\cdot)$, a unary operation $(^{-1})$, and a constant or 0-ary operation e; satisfying the following axioms

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

for any free variables x, y, z. The problem with this notion of algebraic theory is that it is not independent from its representation: there may be multiple formulations for the same theory, with different but equivalent axioms. For example, [McC91] discusses many single-equation axiomatizations of groups, such as

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

with the binary operation /, related to the usual multiplication as $x/y = x \cdot y^{-1}$. Our solution to this problem will be to capture all the algebraic information of a theory – all operations, constants and axioms – into a category. Differently presented but equivalent axioms will give rise to the same category.

Definition 4.1 (Lawvere algebraic theory)**.** An **algebraic theory** [AB17] is a category $\mathbb{A}$ with all finite products whose objects form a sequence $A^0, A^1, A^2, \dots$ such that $A^m \times A^n = A^{m+n}$ for any m, n.

An algebraic theory can be built from its operations and axioms as follows: objects represent natural numbers, $A^0, A^1, A^2, \dots$, and morphisms from A^n to A^m are given by a tuple of m terms $t_1, \dots, t_m$ depending on n free variables $x_1, \dots, x_n$, written as

$$(x_1 \dots x_n \vdash \langle t_1, \dots, t_k \rangle) \colon A^n \to A^m.$$

Composition is defined as componentwise substitution of the terms of the first morphism into the variables of the second one; that is, given $(x_1 \dots x_k \vdash \langle t_1, \dots, t_m \rangle) \colon A^k \to A^m$ and $(x_1 \dots x_m \vdash \langle u_1, \dots, u_n \rangle) \colon A^m \to A^n$, their composition is $(x_1 \dots x_k \vdash \langle s_1, \dots, s_n \rangle)$, where $s_i = u_i[t_1, \dots, t_m / x_1, \dots, x_m]$. Two morphisms are considered equal, $(x_1 \dots x_n \vdash \langle t_1, \dots, t_k \rangle) = (x_1 \dots x_n \vdash \langle t'_1, \dots, t'_k \rangle)$ when componentwise equality of terms, $t_i = t'_i$, follows from the axioms of the theory. Note that identity is the morphism $(x_1 \dots x_n \vdash \langle x_1, \dots, x_n \rangle)$. The kth-projection from A^n is the term $(x_1 \dots x_n \vdash x_k)$, and it is easy to check that these projections make A^n the n-fold product of A. We have built a category with the desired properties.

Definition 4.2 (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.

The **category of models**, $\mathrm{Mod}_{\mathcal{C}}(\mathbb{A})$, is the subcategory of the functor category $\mathcal{C}^{\mathbb{A}}$ containing the functors that preserve all finite products, with the usual natural transformations between them. We say that a category is *algebraic* if it is equivalent to a category of the form $\mathrm{Mod}_{\mathcal{C}}(\mathbb{A})$.

Example 4.3 (The algebraic theory of groups)*.* Let $\mathbb{G}$ be the algebraic theory of groups built from its axioms; we have all the tuples of terms that can be inductively built with

$$(x, y \vdash x \cdot y) \colon G^2 \to G, \qquad (x \vdash x^{-1}) \colon G \to G, \qquad (\vdash e) \colon G,$$

and the projections $(x_1 \dots x_n \vdash x_k) \colon G^n \to G$, where the usual group axioms hold. A model $H \colon \mathbb{G} \to \mathcal{C}$ is determined by an object of $\mathcal{C}$ and some *morphisms of the catgeory* with the above signature for which the axioms hold; for instance,

- a model of $\mathbb{G}$ in Set is a classical **group**, a set with multiplication and inverse functions for which the axioms hold;
- a model of $\mathbb{G}$ in Top is a **topological group**, a topological space with continuous multiplication and inverse functions;
- a model of $\mathbb{G}$ in Mfd, the category of differentiable manifolds with smooth functions between them, is a **Lie group**;
- a model of $\mathbb{G}$ in Grp is an **abelian group**, by the Eckmann-Hilton argument;
- a model of $\mathbb{G}$ in CRing, the category of commutative rings with homomorphisms, is a **Hopf algebra**.

The category of models $\mathrm{Mod}_{\mathsf{Set}}(\mathbb{A})$ is the usual category of groups, Grp; note that natural transformations on this category are precisely group homomorphisms, as they have to preserve the unit, product and inverse of the group in order to be natural.

By construction we know that, if an equation can be proved from the axioms, it is valid in all models (our semantics are *sound*); but we would like to also prove that, if every model of the theory satisfies a particular equation, it can actually be proved from the axioms of the theory (our semantics are *complete*). In general, we can actually prove a stronger, but maybe unsatisfying, result.

Theorem 4.4 (Universal model)**.** *Given $\mathbb{A}$ an algebraic theory, there exists a category $\mathcal{A}$ with a model $U \in \mathrm{Mod}_{\mathcal{A}}(\mathbb{A})$ such that, for every terms u, v, the equality $u = v$ is true in U if and only if $\mathbb{A}$ proves $u = v$. A category with this property is called a* ***universal model****.*

Proof. Indeed, taking $\mathcal{A} = \mathbb{A}$ as a model of itself with the identity functor $U = \mathrm{Id}$, $u = v$ is satisfied under the identity functor if and only if it is satisfied in the original category. □

This proof feels rather disappointing because this trivial model is not even set-theoretic in general; but we can go further and assert the existence of a universal model in a presheaf category via the Yoneda embedding (Definition 3.31).

Corollary 4.5 (Completeness on presheaves)**.** *The Yoneda embedding $y \colon \mathbb{A} \to \mathsf{Set}^{\mathbb{A}^{op}}$ is a universal model for $\mathbb{A}$.*

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

Example 4.6 (Universal group). For instance, a universal model of the group would be the Yoneda embedding of $\mathbb{G}$ in $\mathsf{Set}^{\mathbb{G}^{op}}$. The group object would be the functor $U = \hom_{\mathbb{G}}(-, A^1)$; which can be thought as a family of sets parametrized over the naturals: for each n we have $U_n = \hom_{\mathbb{G}}(A^n, A^1)$, which is the set of terms on n variables under the axioms of a group. In other words, the universal model for the theory of groups would be the free group on n generators, parametrized over n.

4.2.2 Cartesian closed categories

Definition 4.7 (Cartesian closed category)**.** A **cartesian closed category** is a category $\mathcal{C}$ in which the terminal, diagonal and product functors have right adjoints

$$!\colon \mathcal{C} \to 1, \qquad \Delta\colon \mathcal{C} \times \mathcal{C} \to \mathcal{C}, \qquad (- \times A)\colon \mathcal{C} \to \mathcal{C}.$$

These adjoints are given by terminal, product and exponential objects, written as

$$\frac{* \longrightarrow *}{C \xrightarrow{!} 1} \qquad \frac{C, C \xrightarrow{f,g} A, B}{C \xrightarrow{\langle f,g \rangle} A \times B} \qquad \frac{C \times A \xrightarrow{f} B}{C \xrightarrow{\widetilde{f}} B^A}$$

Exponentials are characterized by the **evaluation morphism** $\varepsilon\colon B^A \times A \to B$ which is the counit of the adjunction

$$\frac{C \times A \xrightarrow{\widetilde{f} \times \mathrm{id}} B^A \times A \xrightarrow{\varepsilon} B \quad (f)}{C \xrightarrow[\widetilde{f}]{} B^A \xrightarrow[\mathrm{id}]{} B^A \quad (\widetilde{f})}$$

Example 4.8 (Presheaf categories are cartesian closed). Set is cartesian closed, with exponentials given by function sets. In general, any presheaf category $\mathsf{Set}^{\mathcal{C}^{op}}$ from a small $\mathcal{C}$ is cartesian closed. Given any two presheaves Q, P, if the exponential exists, its component in any A should be

$$Q^P A \cong \mathrm{Nat}(\hom(-, A), Q^P) \cong \mathrm{Nat}(\hom(-, A) \times P, Q)$$

by the Yoneda lemma, which is a set when $\mathcal{C}$ is small. A family of evaluation functions $\varepsilon_A \colon \mathrm{Nat}(\hom(-, A) \times P, Q) \times PA \to QA$ can be defined as $\varepsilon_A(\eta, p) = \eta(\mathrm{id}_A, p)$. We show that each is a universal arrow: given any $n \colon R \times P \Rightarrow Q$, we can show that there exists a unique ϕ making the diagram

$$\begin{array}{ccc} R \times P & & \\ {\scriptstyle \phi \times \mathrm{Id}} \downarrow & \searrow^{n} & \\ \mathrm{Nat}(\hom(-, A) \times P, Q) \times P & \xrightarrow[\varepsilon]{} & Q \end{array}$$

commute. In fact, we know that, by commutativity $\phi_r(\mathrm{id}, p) = \varepsilon_A(\phi_r, p) = n(r, p)$ must hold; and then by naturality, for every $f \colon B \to A$,

$$\begin{array}{ccc} RA & \xrightarrow{\phi_A} & \mathrm{Nat}(\hom(-, A) \times P, Q) \\ {\scriptstyle Rf}\downarrow & & \downarrow{\scriptstyle -\circ((f\circ -)\times \mathrm{id})} \\ RB & \xrightarrow{\phi_B} & \mathrm{Nat}(\hom(-, B) \times P, Q) \end{array}$$

Thus, the complete natural transformation is completely determined for any $r \in RA$, $p \in PA$ as

$$\phi(r)(f, p) = \phi(Rf(r))(\mathrm{id}, p) = n(Rfr, p).$$

The existence of a universal family of evaluations characterizes the adjunction by Proposition 3.39. In general, cartesian-closed categories with functors preserving products and exponentials form a category called Ccc that we show equivalent to a category containing lambda theories in Theorem 4.14.

4.2.3 Simply-typed λ-theories

If we read $\Gamma \vdash a : A$ as a morphism from the context Γ to the output type A, the rules of simply-typed lambda calculus with product and unit types match the adjoints that determine a cartesian closed category

$$\frac{}{\Gamma \vdash * : 1} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \langle a, b \rangle : A \times B} \qquad \frac{\Gamma, a : A \vdash b : B}{\Gamma \vdash (\lambda a.b) : A \to B}$$

A **λ-theory** $\mathbb{T}$ is the analog of a Lawvere theory for cartesian closed categories. It is given by a set of basic types and constants over the simply-typed lambda calculus and a set of equality axioms, determining a **definitional equality** $\equiv$, an equivalence relation preserving the structure of the simply-typed lambda calculus; that is

$$
\begin{array}{ll}
t \equiv *, & \text{for each } t : 1; \\
\langle a, b\rangle \equiv \langle a', b'\rangle, & \text{for each } a \equiv a',\ b \equiv b' \\
\mathtt{fst}\langle a, b\rangle \equiv a,\ \mathtt{snd}\langle a, b\rangle \equiv b, & \text{for each } a : A, b : B; \\
\mathtt{fst}\ m \equiv \mathtt{fst}\ m',\ \mathtt{snd}\ m \equiv \mathtt{snd}\ m', & \text{for each } m \equiv m'; \\
m \equiv \langle \mathtt{fst}\ m, \mathtt{snd}\ m\rangle, & \text{for each } m : A \times B; \\
f\ x \equiv f'\ x', & \text{for each } f \equiv f', x \equiv x'; \\
(\lambda x.f\ x) \equiv f, & \text{for each } f : A \to B; \\
(\lambda x.m) \equiv (\lambda x.m'), & \text{for each } m \equiv m'; \\
(\lambda x.m)\ n \equiv m[n/x] & \text{for each } m : B, n : A.
\end{array}
$$

Two types are *isomorphic*, $A \cong A'$ if there exist terms $f\colon A \to A'$ and $g\colon A' \to A$ such that $f\ (g\ a) \equiv a$ for each $a{:}A$, and $g\ (f\ a') \equiv a'$ for each $a'{:}A'$.

Example 4.9. Gödel's **System T** [GTL89] is defined as a λ-theory with the basic types $\mathtt{nat}$ and $\mathtt{bool}$; the constants $0 : \mathtt{nat}$, $S : \mathtt{nat} \to \mathtt{nat}$, $\mathrm{true} : \mathtt{bool}$, $\mathrm{false} : \mathtt{bool}$, $\mathrm{ifelse} : \mathtt{bool} \to C \to C \to C$ and $\mathrm{rec} : C \to (\mathtt{nat} \to C \to C) \to \mathtt{nat} \to C$ where C is any type; and the axioms

$$
\begin{array}{ll}
\mathrm{ifelse}\ \mathrm{true}\ a\ b \equiv a, & \mathrm{rec}\ c_0\ c_s\ 0 \equiv c_0, \\
\mathrm{ifelse}\ \mathrm{false}\ a\ b \equiv b, & \mathrm{rec}\ c_0\ c_s\ (Sn) \equiv c_s\ n\ (\mathrm{rec}\ c_0\ c_s\ n).
\end{array}
$$

Example 4.10 (Untyped $\lambda - calculus$). Untyped λ calculus can be recovered as a λ theory with a single basic type D and a type isomorphism $\mathsf{D} \cong \mathsf{D} \to \mathsf{D}$ given by two constants $r : \mathsf{D} \to (\mathsf{D} \to \mathsf{D})$ and $s : (\mathsf{D} \to \mathsf{D}) \to \mathsf{D}$ such that $r\ (s\ f) \equiv f$ for each $f : \mathsf{D} \to \mathsf{D}$ and $s\ (r\ x) \equiv x$ for each $x : \mathsf{D}$. We assume that each term of the untyped calculus is of type D and apply r, s as needed to construct well-typed terms.

Definition 4.11. The reasonable notion of homomorphism between lambda theories is called a **translation** between λ-theories. Given two lambda theories $\mathbb{T}$ and $\mathbb{U}$, a translation $\tau\colon \mathbb{T} \to \mathbb{U}$ is given by a function both on types and terms that

1. preserves type constructors
$$\tau 1 = 1, \qquad \tau(A \times B) = \tau A \times \tau B, \qquad \tau(A \to B) = \tau A \to \tau B;$$
2. preserves the term structure
$$\begin{array}{lll} \tau(\mathtt{fst}\ m) \equiv \mathtt{fst}\ (\tau m), & \tau(\mathtt{snd}\ m) \equiv \mathtt{snd}\ (\tau m), & \tau\langle a, b\rangle \equiv \langle \tau a, \tau b\rangle, \\ \tau(f\ x) \equiv (\tau f)\ (\tau x), & \tau(\lambda x.m) \equiv \lambda x.(\tau m); & \end{array}$$
3. and preserves all equations, meaning that $t \equiv u$ implies $\tau t \equiv \tau u$.

We consider the category $\lambda\mathsf{Thr}$ of λ-theories with translations as morphisms. Note that the identity is a translation and that the composition of two translations is again a translation. Our goal is to prove that this category is equivalent to that of cartesian closed categories with functors preserving products and exponentials.

Apart from the natural definition of isomorphism, we consider the weaker notion of *equivalence of theories*. Two theories with translations $\tau\colon \mathbb{T} \to \mathbb{U}$ and $\sigma\colon \mathbb{U} \to \mathbb{T}$ are **equivalent** $\mathbb{T} \simeq \mathbb{U}$ if there exist two families of *type isomorphisms* $\tau\sigma A \cong A$ and $\sigma\tau B \cong B$ parametrized over the types A and B.

4.2.4 Syntactic categories and internal languages

Proposition 4.12. *Given a λ-theory $\mathbb{T}$, its **syntactic category** $\mathcal{S}(\mathbb{T})$, has an object for each type of the theory and a morphism $A \to B$ for each term $a : A \vdash b : B$. The composition of two morphisms $a : A \vdash b : B$ and $b' : B \vdash c : C$ is given by $a{:}A \vdash c[b/b']{:}C$; and any two morphisms $\Gamma \vdash b : B$ and $\Gamma \vdash b' : A$ are equal if $b \equiv b'$.*

The syntactic category is cartesian closed and this induces a functor $\mathcal{S}\colon \lambda\mathsf{Thr} \to \mathsf{Ccc}$.

Proof. The type 1 is terminal because every morphism $\Gamma \vdash t{:}1$ is $t \equiv *$. The type $A \times B$ is the product of A and B; projections from a pair morphism are given by $\mathtt{fst}\ \langle a, b\rangle \equiv a$ and $\mathtt{snd}\ \langle a, b\rangle \equiv b$. Any other morphism under the same conditions must be again the pair morphism because $d \equiv \langle \mathtt{fst}\ d, \mathtt{snd}\ d\rangle \equiv \langle a, b\rangle$.

Finally, given two types A, B, its exponential is $A \to B$ with the evaluation morphism

$$m : (A \to B) \times A \vdash (\mathtt{fst}\ m)\ (\mathtt{snd}\ m) : B.$$

It is universal: for any $p{:}C \times A \vdash q{:}B$, there exists a morphism $z{:}C \vdash \lambda x.q[\langle z, x\rangle/p] : A \to B$ such that

$$(\lambda x.q[\langle \mathtt{fst}\ p, x\rangle/p])(\mathtt{snd}\ p) \equiv q[\langle \mathtt{fst}\ p, \mathtt{snd}\ p\rangle/p] \equiv q[p/p] \equiv q;$$

and if any other morphism $z{:}C \vdash d{:}A \to B$ satisfies $(d[\mathtt{fst}\ p/z](\mathtt{snd}\ p)) \equiv q$ then

$$\lambda x.q[\langle z, x\rangle/p] \equiv \lambda x.(d[\mathtt{fst}\ p/z]\ (\mathtt{snd}\ p))[\langle z, x\rangle/p] \equiv \lambda x.(d[z/z]\ x) \equiv d.$$

Given a translation $\tau\colon \mathbb{T} \to \mathbb{U}$, we define a functor $\mathcal{S}(\tau)\colon \mathcal{S}(\mathbb{T}) \to \mathcal{S}(\mathbb{U})$ mapping the object $A \in \mathcal{S}(\mathbb{T})$ to $\tau A \in \mathcal{S}(\mathbb{U})$ and any morphism $\vdash b : B$ to $\vdash \tau b : \tau B$. The complete structure of the functor is then determined because it must preserve products and exponentials. □

Proposition 4.13 (Internal language)**.** *Given a cartesian closed category $\mathcal{C}$, its **internal language** $\mathbb{L}(\mathcal{C})$ is a λ-theory with a type $\ulcorner A \urcorner$ for each object $A \in \mathcal{C}$, a constant $\ulcorner f \urcorner\colon \ulcorner A \urcorner \to \ulcorner B \urcorner$ for each morphism $f\colon A \to B$, axioms*

$$\ulcorner \mathrm{id} \urcorner\ x \equiv x, \qquad\qquad \ulcorner g \circ f \urcorner\ x \equiv \ulcorner g \urcorner\ (\ulcorner f \urcorner\ x),$$

and three families of constants

$$\mathtt{T} : 1 \to \ulcorner 1 \urcorner, \quad \mathtt{P}_{AB} : \ulcorner A \urcorner \times \ulcorner B \urcorner \to \ulcorner A \times B \urcorner, \quad \mathtt{E}_{AB} : (\ulcorner A \urcorner \to \ulcorner B \urcorner) \to \ulcorner B^A \urcorner,$$

that act as type isomorphisms*, which means that they create the following pairs of two-side inverses relating the categorical and type-theoretical structures*

$$\begin{array}{ll}
t \equiv \mathtt{T}\ * & \text{for each } u : \ulcorner 1 \urcorner, \\
m \equiv \mathtt{P}\ \langle \ulcorner \pi_1 \urcorner\ m, \ulcorner \pi_2 \urcorner\ m\rangle & \text{for each } z : \ulcorner A \times B \urcorner, \\
n \equiv \langle \ulcorner \pi_0 \urcorner\ (\mathtt{P}\ n), \ulcorner \pi_1 \urcorner\ (\mathtt{P}\ n)\rangle & \text{for each } n : \ulcorner A \urcorner \times \ulcorner B \urcorner, \\
f \equiv \mathtt{E}\ (\lambda x.\ulcorner e \urcorner\ (\mathtt{P}\ \langle f, x\rangle)) & \text{for each } f : \ulcorner B^A \urcorner, \\
g \equiv \lambda x.\ulcorner e \urcorner\ (\mathtt{P}\ \langle \mathtt{E}\ g, x\rangle) & \text{for each } g : \ulcorner A \urcorner \to \ulcorner B \urcorner.
\end{array}$$

This extends to a functor $\mathbb{L}\colon \mathsf{Ccc} \to \lambda\mathsf{Thr}$.

Proof. Given any functor preserving products and exponentials $F\colon \mathcal{C} \to \mathcal{D}$, we define a translation $\mathbb{L}(F)\colon \mathbb{L}(\mathcal{C}) \to \mathbb{L}(\mathcal{D})$ taking each basic type $\ulcorner A \urcorner$ to $\ulcorner FA \urcorner$ and each constant $\ulcorner f \urcorner$ to $\ulcorner Ff \urcorner$; equations are preserved because F is a functor and types are preserved up to isomorphism because F preserves products and exponentials. □

Theorem 4.14 (Equivalence between cartesian closed categories and lambda calculus)**.** *There exists a equivalence of categories* $\mathcal{C} \simeq \mathcal{S}(\mathbb{L}(\mathcal{C}))$ *for any* $\mathcal{C} \in \mathsf{Ccc}$ *and an equivalence of theories* $\mathbb{T} \simeq \mathbb{L}(\mathcal{S}(\mathbb{T}))$ *for any* $\mathbb{T} \in \lambda\mathsf{Thr}$*.*

Proof. On the one hand, we define $\eta \colon \mathcal{C} \to \mathcal{S}(\mathbb{L}(\mathcal{C}))$ as $\eta A = \ulcorner A \urcorner$ in objects and $\eta f = (a : \ulcorner A \urcorner \vdash f\ a : \ulcorner B \urcorner)$ for any morphism $f \colon A \to B$. It is a functor because $\ulcorner \mathrm{id} \urcorner\ a \equiv a$ and $\ulcorner g \circ f \urcorner\ a \equiv g\ (f\ a)$. We define $\theta : \mathcal{S}(\mathbb{L}(\mathcal{C})) \to \mathcal{C}$ on types inductively as $\theta(1) = 1$, $\theta(\ulcorner A \urcorner) = A$, $\theta(B \times C) = \theta(A) \times \theta(C)$ and $\theta(B \to C) = \theta(C)^{\theta(B)}$. Now, there is a natural isomorphism $\eta\theta \Rightarrow \mathrm{Id}$, using the isomorphisms induced by the constants $\mathtt{T}, \mathtt{P}, \mathtt{E}$,

$$\begin{aligned}
&\eta(\theta \ulcorner A \urcorner) = \eta A = \ulcorner A \urcorner,\\
&\eta(\theta 1) = \eta 1 = \ulcorner 1 \urcorner \cong 1,\\
&\eta(\theta(A \times B)) = \ulcorner \theta(A) \times \theta(B) \urcorner \cong \ulcorner \theta A \urcorner \times \ulcorner \theta B \urcorner = \eta\theta A \times \eta\theta B \cong A \times B.\\
&\eta(\theta(A \to B)) = \ulcorner \theta(A) \to \theta(B) \urcorner = \ulcorner \theta B \urcorner^{\ulcorner \theta A \urcorner} = (\eta\theta B)^{(\eta\theta A)} = B^A;
\end{aligned}$$

and a natural isomorphism $\mathrm{Id} \Rightarrow \theta\eta$ which is in fact an identity, $A = \theta \ulcorner A \urcorner = \theta\eta(A)$.

On the other hand, we define a translation $\tau \colon \mathbb{T} \to \mathbb{L}(\mathcal{S}(\mathbb{T}))$ as $\tau A = \ulcorner A \urcorner$ in types and $\tau(a) = \ulcorner (\vdash a : \tau A) \urcorner$ in constants. We define $\sigma \colon \mathbb{L}(\mathcal{S}(\mathbb{T})) \to \mathbb{T}$ as $\sigma \ulcorner A \urcorner = A$ in types and as

$$\sigma(\ulcorner a : A \vdash b : B \urcorner) = \lambda a.b, \quad \sigma\mathtt{T} = \lambda x.x, \quad \sigma\mathtt{P} = \lambda x.x, \quad \sigma\mathtt{E} = \lambda x.x,$$

in the constants of the internal language. We have $\sigma(\tau(A)) = A$, so we will check that $\tau(\sigma(A)) \cong A$ by structural induction on the constructors of the type:

- if $A = \ulcorner B \urcorner$ is a basic type, we apply structural induction over the type B to get
  - if B is a basic type, $\tau\sigma(\ulcorner B \urcorner) = \ulcorner B \urcorner$;
  - if $B = 1$, then $\tau\sigma(\ulcorner 1 \urcorner) = 1$ and $\ulcorner 1 \urcorner \cong 1$ thanks to the constant $\mathtt{T}$;
  - if $B = C \times D$, then $\tau\sigma(\ulcorner C \times D \urcorner) = \ulcorner C \urcorner \times \ulcorner D \urcorner$ and $\ulcorner C \times D \urcorner \cong \ulcorner C \urcorner \times \ulcorner D \urcorner$ thanks to the constant $\mathtt{P}$;
  - if $B = D^C$, then $\tau\sigma(\ulcorner D^C \urcorner) = \ulcorner C \urcorner \to \ulcorner D \urcorner$ and $\ulcorner D^C \urcorner \cong \ulcorner C \urcorner \to \ulcorner D \urcorner$ thanks to the constant $\mathtt{E}$.
- if $A = 1$, then $\tau\sigma 1 = 1$;
- if $A = C \times D$, then $\tau\sigma(C \times D) = \tau\sigma(C) \times \tau\sigma(D) \cong C \times D$ by induction hypothesis;
- if $A = C \to D$, then $\tau\sigma(C \to D) = \tau\sigma(C) \to \tau\sigma(D) \cong C \to D$ by induction hypothesis.

□

Thus, we can say that the simply-typed lambda calculus is the internal language of cartesian closed categories; each lambda theories are cartesian closed categories.

4.3 Working in cartesian closed categories

4.3.1 Diagonal arguments

We can now talk internally about cartesian closed categories using lambda calculus. Note each closed λ term $\vdash a : A$ can also be seen as a morphism from the terminal object $1 \to A$. We use this language to provide a simple proof to a known theorem for catesian closed categories

by W. Lawvere (see [Lawa] for details). The theorem will imply many known corollaries as particular cases when interpreted in different contexts.

Theorem 4.15 (Lawvere, 1969)**.** *We say that a morphism $g : A \to B$ in a cartesian closed category is* ***point-surjective*** *if, for every element $b : B$, there exists an element $a : A$ such that $g\ a \equiv b$. In any cartesian closed category, if there exists a point-surjective morphism $d : A \to B^A$, then each morphism $f : B \to B$ has a fixed point $b : B$, such that $f\ b \equiv b$.*

Proof. As d is point-surjective, there exists $x : A$ such that $d\ x \equiv \lambda a.f\ (d\ a\ a)$, but then, $d\ x\ x \equiv (\lambda a.f\ (d\ a\ a))\ x \equiv f\ (d\ x\ x)$ is a fixed point. □

Corollary 4.16 (Cantor, 1878)**.** *Let A be a set. The set of all subsets of A has a strictly greater cardinality than A.*

Proof. Every subset of A is uniquely determined by a function to the set of two elements $A \to 2$. As there exists nontrivial permutation of the two-element set, a point-surjective $A \to 2^A$ cannot exist by virtue of Theorem 4.15. □

Corollary 4.17 (Russell, 1901)**.** *In a naive formulation of set theory, every collection is a set. This leads to inconsistency.*

Proof. We could consider Sets, the class of all sets, and the membership relation $\in : \mathsf{Sets} \to 2^{\mathsf{Sets}}$. This relation would be point-surjective if we assume that for any property on the class of sets, given as a morphism $P : \mathsf{Sets} \to 2$, there exists a comprehension set $\{y \in \mathsf{Sets} \mid P(y)\}$. In that case, again, any permutation of the two-element set would have a fixed point, which is false. □

Corollary 4.18. *Every term in untyped λ-calculus has a fixed point.*

Proof. We consider untyped λ-calculus as a theory (Example 4.10) where terms can be regarded as morphisms $D \to D$. There exists a type isomorphism $D \to (D \to D)$, which is in particular a point-surjection. Thus, by Theorem 4.15, there exists a fixed point for any term $D \to D$. Note that the term thus constructed is precisely the fixed point we defined in Section 2.5.7. □

Corollary 4.19 (Gödel, Tarski, 1936)**.** *A consistent theory cannot express its own truth. In particular, no consistent formal system of arithmetic can encode the truth of arithmetic statements.*

Proof. Let our category be a theory (in the sense of Definition 4.1) with objects $A^0, A^1, A^2, \ldots$ and a supplementary object 2. We say that the theory is *consistent* if there exists a morphism $\mathrm{not} : 2 \to 2$ such that $\mathrm{not} \circ \varphi \neq \varphi$ for every $\varphi : A \to 2$. We say that *truth* is definable if there exists a map $\mathrm{truth} : A \times A \to 2$ such that for every predicate $\varphi : A \to 2$, there exists a *Gödel number* $c : A$ such that $\mathrm{truth}(c, a) = \varphi(a)$. By Theorem 4.15 we know that if satisfiability is definable in a theory then it must be inconsistent (see [Yan03]). □

We want to note here how abstracting a seemingly banal theorem has resulted in a myriad of deep results. Moreover, Lawvere's original theorem can be replicated without exponentials in any category with all products, taking adjoints in $d : A \times A \to B$. The condition of d being point-surjective can also be weakened; we only need, for every $g : A \to B$, the existence of some $x : A$ such that $d\ x\ a \equiv g\ a$ for all $a : A$.

4.3.2 Bicartesian closed categories

Definition 4.20. A **bicartesian closed category** is a cartesian closed category in which the terminal and diagonal functors have left adjoints. These adjoints are given by initial and coproduct objects, written as inference rules as follows.

$$\frac{* \longrightarrow *}{0 \xrightarrow{!} C} \qquad \frac{A, B \xrightarrow{f,g} C, C}{A + B \xrightarrow{f+g} C}$$

The rules of union and void types in simply-typed lambda calculus can be rewritten to match the structure of these adjoints.

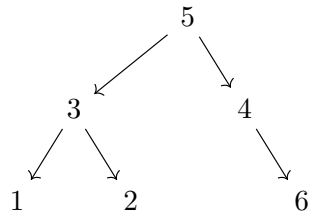
$$\frac{}{\Gamma, z : 0 \vdash \text{abort} : C} \qquad \frac{\Gamma, a : A \vdash c : C \qquad \Gamma, b : B \vdash c' : C}{\Gamma, u : A + B \vdash \text{case } u \text{ of } c; c' : C}$$

In a similar way to how our previous fragment of simply-typed lambda calculus was the internal language of cartesian closed categories, the extended version of lambda calculus defined in Section 1.3.1 is the internal language of bicartesian closed categories.

4.3.3 Inductive types

The existence of an initial algebra in a cartesian closed category is equivalent to the existence a type with an elimination rule representing an induction principle. Types generated by an inductive rule are called **inductive types** and are common constructs in functional programming languages. Inductive types and their properties can be encoded as initial algebras satisfying a certain universal property. In particular, we will see examples of inductive types and how all of them can be modeled as initial algebras over a certain class of functors.

Example 4.21 (Binary tree data structure). Let T be a type generated by the constructors $\mathsf{nil} : T$ and $\mathsf{node} : \mathbb{N} \to T \to T \to T$. It can be seen as the data structure of a binary tree of naturals, where "nil" is an empty leaf and "node" builds a data structure with a natural number on top and two pointers to binary trees.



For instance, the above diagram represents the term

$$\mathsf{node}\ 5\ (\mathsf{node}\ 3\ (\mathsf{node}\ 1\ \mathsf{nil}\ \mathsf{nil})\ (\mathsf{node}\ 2\ \mathsf{nil}\ \mathsf{nil}))\ (\mathsf{node}\ 4\ \mathsf{nil}\ (\mathsf{node}\ 6\ \mathsf{nil}\ \mathsf{nil})).$$

Example 4.22 (Semantics of System T). System T was described in Example 4.9 as an extension of the simply typed lambda calculus with natural numbers and booleans. A cartesian closed category with a natural numbers object given as an initial algebra (Example 3.46) has an internal language with a type $\mathbb{N}$ and an elimination principle ind given by the universal property of the algebra, as in the following diagram.

$$\begin{array}{ccc} 1+\mathbb{N} & \longrightarrow & 1+C \\ {\scriptstyle\langle 0,\mathrm{succ}\rangle}\downarrow & & \downarrow{\scriptstyle\langle x,f\rangle} \\ \mathbb{N} & \xrightarrow{\mathsf{ind}\ x\ f} & C \end{array} \qquad \begin{array}{c} \mathsf{ind}\colon C \to (C \to C) \to (\mathbb{N} \to C) \\ \mathsf{ind}\ x\ f\ 0 \equiv x \\ \mathsf{ind}\ x\ f\ (S\ n) \equiv f\ (\mathsf{ind}\ x\ f\ n) \end{array}$$

It could seem that this induction principle is weaker than the recursion principle System T offered, $\mathsf{rec} : D \to (\mathbb{N} \to D \to D) \to \mathbb{N} \to D$. However, the full recursion principle can be recovered using the cartesian closed structure and taking $C = \mathbb{N} \times D$ when applying the induction principle to define

$$\mathsf{rec}\ c_0\ c_s\ n \equiv \pi_1\ (\mathsf{ind}\ \langle c_0, 0\rangle\ (\lambda m.\ \langle c_s\ m\ d, \mathrm{succ}\ m\rangle)\ n).$$

It can be checked that rec is such that $\mathsf{rec}\ c_0\ c_s\ 0 \equiv c_0$, and $\mathsf{rec}\ c_0\ c_s\ (S\ n) \equiv c_s\ n\ (\mathsf{rec}\ c_0\ c_s\ n)$.

Finally, note that ind can be directly recovered from rec, and the booleans of System T correspond to the coproduct $1 + 1$. System T is thus equivalent to the internal language of a cartesian closed category with a natural numbers object and a coproduct $1+1$; and it can take models in any category with this structure.

Example 4.23 (Polynomial endofunctors). Following [Awo10] and [MP00], we can generalize these examples to initial algebras over **polynomial endofunctors** of the form

$$P(X) = C_0 + C_1 \times X + C_2 \times X^2 + \cdots + C_n \times X^n$$

for some fixed objects $C_0, \ldots, C_n$. Where the initial algebra can be seen to correspond to an inductive branching type with n-ary nodes of type C_n and leafs of type C_0. Natural numbers $(1 + X)$ and lists $(1 + A \times X)$ for a fixed type A, are particular examples of initial algebras of polynomial endofunctors (as seen in Examples 3.46 and 3.47). Binary trees, as described in Example 4.21, are the initial algebra for the functor $T \mapsto (1 + \mathbb{N} \times T \times T)$.

4.4 Locally cartesian closed categories and dependent types

4.4.1 Quantifiers and subsets

The motivation for this section is the interpretation of logical formulas as subsets. Every predicate P on a set A can be seen as a subset $\{a \in A \mid P(a)\} \hookrightarrow A$ determined by the inclusion monomorphism. Under this interpretation, logical implication between two propositions, $P \to Q$, can be seen as a morphism that commutes with the two inclusions; that is,

$$\begin{array}{ccc} \{a \in A \mid P(a)\} & \longrightarrow & \{a \in A \mid Q(a)\} \\ & \searrow \quad \swarrow & \\ & A & \end{array}$$

P implies Q if and only if each $a \in A$ such that $P(a)$ is also in the subset of elements such that $Q(a)$. Note how we are working in a subcategory of the slice category Set/A.

Given a function $f\colon A \to B$, any property in B induces a property on A via **substitution** in the relevant logical formula. This substitution can be encoded categorically as a pullback: the

pullback of a proposition along a function is the proposition induced by substitution.

$$\begin{array}{ccc} \{a \in A \mid P(f(a))\} & \longrightarrow & \{b \in B \mid P(b)\} \\ \downarrow & & \downarrow \\ A & \xrightarrow{\quad f \quad} & B \end{array}$$

A particular case of a substitution is **logical weakening**: a proposition on the set A can be seen as a proposition on $A \times B$ where we simply discard the B component of a pair.

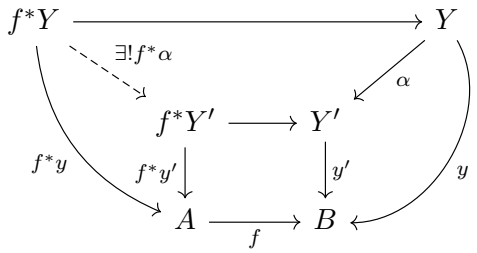
$$\begin{array}{ccc} \{(a,b) \in A \times B \mid P(\pi(a,b))\} & \longrightarrow & \{a \in A \mid P(a)\} \\ \downarrow & & \downarrow \\ A \times B & \xrightarrow{\quad \pi \quad} & A \end{array}$$

Although it seems like an uninteresting particular case, once we formalize this operation as a functor, existential and universal quantifiers can be obtained as adjoints to weakening.

Definition 4.24 (The pullback functor)**.** Given a function $f\colon A \to B$ in any category $\mathcal{C}$ with all pullbacks, the **pullback functor** $f^*\colon \mathcal{C}/B \to \mathcal{C}/A$ is defined for any object $y\colon Y \to B$ as the object $f^*y\colon (f^*Y) \to A$ such that

$$\begin{array}{ccc} (f^*Y) & \longrightarrow & Y \\ {\scriptstyle f^*y}\downarrow & & \downarrow{\scriptstyle y} \\ A & \xrightarrow{\quad f \quad} & B \end{array}$$

is a pullback square. The functor is defined on any morphism $\alpha\colon y \to y'$ between two objects $y\colon Y \to B$, $y'\colon Y' \to B$ as the only morphism making the following diagram commute.



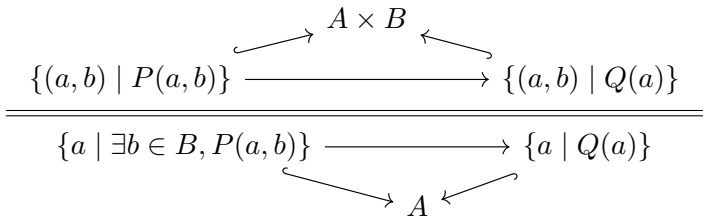
Note that the pullback functor is only defined up to isomorphism in objects and well-defined on morphisms by virtue of the universal property of pullbacks.

In Set, we can find two adjoints to the particular case of the weakening functor $\pi^*\colon \mathcal{C}/A \to \mathcal{C}/(A \times B)$. These two adjoints are $\exists \dashv \pi^* \dashv \forall$ because

- proving the implication $P(\pi(a,b)) \to Q(a,b)$ for each pair (a,b) amounts to prove that $P(a) \to (\forall b \in B, Q(a,b))$ for each a;

$$\begin{array}{ccc} & A \times B & \\ \{(a,b) \mid P(a)\} & \longrightarrow & \{(a,b) \mid Q(a,b)\} \\ \hline\hline \{a \mid P(a)\} & \longrightarrow & \{a \mid \forall b \in B, Q(a,b)\} \\ & A & \end{array}$$

- and proving that $(\exists b \in B, P(a,b)) \to Q(a)$ for each a is the same as proving that $P(a,b) \to Q(\pi(a,b))$ for each pair (a,b).

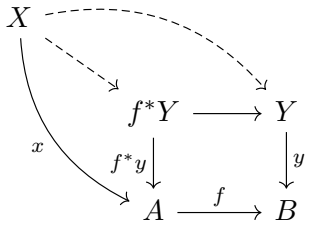


Note how, in this case, we are considering adjunction diagrams in the slice category. A generalization of this idea to other categories will extend our categorical logic with quantifiers.

4.4.2 Locally cartesian closed categories

Proposition 4.25 (Left adjoint of the pullback functor)**.** *Given any category $\mathcal{C}$ with all finite limits and $f\colon A \to B$, the pullback functor $f^*\colon \mathcal{C}/A \to \mathcal{C}/B$ has a left adjoint $\Sigma_f\colon \mathcal{C}/B \to \mathcal{C}/A$ defined as $\Sigma_f x = f \circ x$ in objects and trivially in morphisms.*

Proof. We must find a natural bijection $\hom(f \circ x, y) \cong \hom(x, f^*y)$; but, precisely by the universal property of the pullback, we have a natural bijection between arrows $k\colon X \to f^*Y$ such that $x = f^*y \circ k$ and arrows $\widetilde{k}\colon X \to Y$ such that $f \circ x = y \circ \widetilde{k}$.

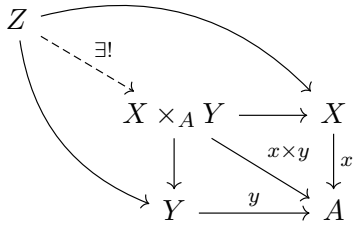


□

We define a **locally cartesian closed category** as a category with a terminal object and pullbacks $\mathcal{C}$ such that the pullback functor also has a right adjoint $\Pi_f\colon \mathcal{C}/A \to \mathcal{C}/B$. The rationale for this name becomes apparent in the following characterization.

Theorem 4.26 (Characterization)**.** *A category $\mathcal{C}$ with terminal object is locally cartesian closed if and only if $\mathcal{C}/A$ is cartesian closed for any object A (see [New17]).*

Proof. ($\Rightarrow$) Suppose $\mathcal{C}$ be locally cartesian closed. The terminal object of $\mathcal{C}/A$ is trivially $\mathrm{id}_A\colon A \to A$, and the product of $x\colon X \to A$ and $y\colon Y \to A$ is given by universal property of the pullback



We can notice that multiplying by $- \times y$ is the same as composing $\Sigma_y \circ y^*\colon \mathcal{C}/A \to \mathcal{C}/A$; and, as we have $\Sigma_y \dashv y^* \dashv \Pi_y$, for any $a, b\colon Z \to A$,

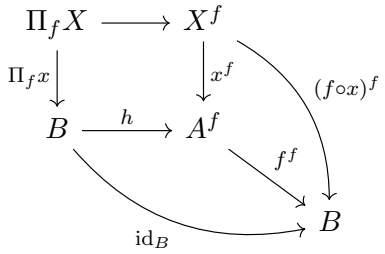
$$\dfrac{\dfrac{\Sigma_y(y^*a) \longrightarrow b}{y^*a \longrightarrow y^*b}}{a \longrightarrow \Pi_y(y^*b)}$$

the product has a right adjoint and the exponential is given by $a^y = \Pi_y y^* a$.

($\Leftarrow$) Suppose $\mathcal{C}$ such that $\mathcal{C}/A$ is always cartesian closed. In particular the slice on the terminal object is cartesian closed and so is $\mathcal{C} \cong \mathcal{C}/1$; we only need to prove the existence of pullbacks in $\mathcal{C}$ and a right adjoint for each pullback functor f^*.

Again, if $f\colon X \to A$ and $g\colon Y \to A$ are objects in the slice category $\mathcal{C}/A$, their product creates a morphism $X \times_A Y \to A$ in $\mathcal{C}$ and the universal property of the product in the slice category is exactly the universal property of the pullback in $\mathcal{C}$.

Now, given $f\colon A \to B$, we will define Π_f. As $\mathcal{C}/B$ is cartesian closed, there is a functor $(-)^f$ that we can apply to any $x\colon X \to A$ when seen as the triangle $x\colon (f \circ x) \to x$ in the slice category. Moreover, the identity $\mathrm{id}_f\colon f \to f$ has a transpose $h\colon \mathrm{id}_B \to f^f$; so we can compute the following pullback on $\mathcal{C}/B$ that defines $\Pi_f x$ as $h^*(x^f)$.

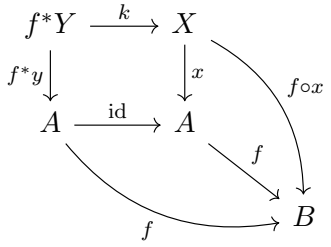


Note that Π_f is defined in objects as the composition of two functors, thus, it can be directly extended to morphisms. We only have to prove that there is a natural bijection $\hom(f^*y, x) \cong \hom(y, \Pi_f x)$.

By the universal property of the pullback, each $k\colon y \to \Pi_f x$ determines two pullback projections $k_1\colon y \to (f \circ x)^f$ and $k_2\colon y \to \mathrm{id}$ such that $x^f \circ k_1 = h \circ k_2$. Applying the adjunction, in both sides of the equation we see that they are determined by morphisms $j\colon f^*Y \to X$ such that $f^*y = x \circ j$.

$$\dfrac{y \xrightarrow{k_2} \mathrm{id} \xrightarrow{h} f^f}{f \times y \xrightarrow{f^*y} f \xrightarrow{\mathrm{id}} f} \qquad \dfrac{y \xrightarrow{k_1} (f \circ x)^f \xrightarrow{x^f} f^f}{f \times y \xrightarrow{j} f \circ x \xrightarrow{x} f}$$

But those morphisms are precisely morphisms $f^*y \to x$ in $\mathcal{C}/A$, and we have established the natural bijection.



□

We proved earlier that "generalized sets", or presheaves, were cartesian closed; we will now prove that they are actually locally cartesian closed.

Theorem 4.27 (Presheaf categories are locally cartesian closed)**.** *Any presheaf category* $\mathsf{Set}^{\mathcal{C}^{op}}$ *from a small category* $\mathcal{C}$ *is locally cartesian closed (see [Awo10]).*

Proof. We will prove that given any $A \in \mathsf{Set}^{\mathcal{C}^{op}}$, there exists a small category $\mathcal{D}$ such that there is an equivalence of categories $\mathsf{Set}^{\mathcal{D}^{op}} \simeq \mathsf{Set}^{\mathcal{C}^{op}}/A$. Thus, every slice is cartesian closed as shown in Example 4.8 and by the characterization of Theorem 4.26, the whole category is locally cartesian closed.

We take $\mathcal{D}$ as a particular case of a comma category where objects are arrows $f\colon yC \to A$ in $\mathsf{Set}^{\mathcal{C}^{op}}$ from the Yoneda embedding of an object $C \in \mathcal{C}$ to the fixed object A. Morphisms are commutative triangles

$$\begin{array}{ccc} yC & \xrightarrow{\varphi} & yC' \\ & {\scriptstyle f}\searrow \quad \swarrow{\scriptstyle f'} & \\ & A & \end{array}$$

where, as Yoneda is a full and faithful embedding, φ must be of the form $\varphi = yh$ for a unique $h\colon C \to C'$. Note how $\mathcal{D}$ can be fully and faithfully embedded inside $\mathsf{Set}^{\mathcal{C}^{op}}/A$ with a functor $I\colon \mathcal{D} \to \mathsf{Set}^{\mathcal{C}^{op}}/A$.

A functor $\Phi\colon \mathsf{Set}^{\mathcal{C}^{op}}/A \to \mathsf{Set}^{\mathcal{D}^{op}}$ can be now defined on objects as $\Phi(q) = \hom_{\mathsf{Set}^{\mathcal{C}^{op}}/A}(I(-), q)$, which is a composition of functors. It remains to show that this functor in fact determines an equivalence. □

4.4.3 Dependent types

In the same way that cartesian closed categories model the types of the simply typed lambda calculus, locally cartesian closed categories model **dependent types** that can depend on elements of another type. Each dependent type B depending on values of type A can be also seen as a family of types parametrized over the other type $\{B(a)\}_{a:A}$. This extension of type theory forces us to revisit the notion of typing context.

Typing contexts for dependent type theory are given as a list of variables

$$\Gamma = (a_1 : A_1, a_2 : A_2, \ldots, a_n : A_n)$$

where each type A_i can depend on the variables $a_1, \ldots, a_{i-1}$. The core syntax of dependent type theory can be expressed in terms of **substitutions** between contexts. A substitution from a context Δ to Γ is written as $\sigma\colon \Delta \to \Gamma$, and is given by a list of terms $(t_1, \ldots, t_n)$ such that

$$\Delta \vdash t_1 : A_1, \quad \Delta \vdash t_2 : A_2[t_1/a_1], \quad \ldots, \quad \Delta \vdash t_n : A_n[t_1, \ldots, t_{n-1}/a_1, \ldots, a_{n-1}],$$

that is, a context can be substituted into another if the list of terms of the second one can be built from the first one. The interpretation of a dependent type theory as a category takes contexts Γ as objects $[\![\Gamma]\!]$ and substitutions as morphisms. Note how there exists an identity substitution $\sigma_{\mathrm{id}}\colon \Gamma \to \Gamma$ that simply lists the variables of the context and how any two substitutions $\tau\colon \Gamma \to \Phi$ and $\sigma\colon \Delta \to \Gamma$ can be composed into $\tau \circ \sigma\colon \Delta \to \Phi$, which creates the terms of Γ from Δ following τ and uses again these terms to create the terms of Φ.

A particular kind of substitutions will be **display maps**. If a term can be constructed on a given context, $\Gamma \vdash a : A$, the context can be extended with that term to $\Gamma, a : A$. Display maps are substitutions of the form $\pi_A \colon [\![\Gamma, a : A]\!] \to [\![\Gamma]\!]$ that simply list the variables of Γ and discard $a : A$.

This way, each type A in a context Γ is represented by the object $\pi_A \colon [\![\Gamma, a : A]\!] \to [\![\Gamma]\!]$ of the slice category $\mathcal{C}/[\![\Gamma]\!]$; and each term of the type, $\Gamma \vdash a : A$ is represented by a morphism from $\mathrm{id} \colon [\![\Gamma]\!] \to [\![\Gamma]\!]$, which is the terminal object of Γ/A, as in the following diagram.

$$\begin{array}{ccc} [\![\Gamma]\!] & \xrightarrow{a} & [\![\Gamma, a : A]\!] \\ & \searrow^{=} & \downarrow{\scriptstyle \pi_A} \\ & & [\![\Gamma]\!] \end{array}$$

4.4.4 Dependent pairs

The locally cartesian closed structure of a category induces new type constructors in the type theory: dependent pairs and dependent functions. Their meaning under the Curry-Howard interpretation is that of the existential and universal quantifiers, respectively.

Dependent pair types, or **Σ-types**, can be seen as a generalized version of product types. Given a family of types parametrized by another type, $\{B(a)\}_{a:A}$, the elements of $\sum_{a:A} B(a)$ are pairs $\langle a, b\rangle$ with a first element $a : A$ and a second element $b : B(a)$; that is, the type of the second component depends on the first component. This type is often written as $\Sigma(a : A), B(a)$ and it corresponds to the intuitionistic existential quantifier under the *propositions as types* interpretation. That is, the proof of $\exists(a : A), B(a)$ must be seen as a pair given by an element a and a proof of $B(a)$.

In a locally closed cartesian category, a type B depending on $\Gamma, a : A$, can be written as $\pi_B \colon [\![\Gamma, a : A, b : B]\!] \to [\![\Gamma, a : A]\!]$; then, the type $\sum_{a:A} B$ is given by the object

$$\Sigma_{\pi_A} \pi_B \colon [\![\Gamma, a : A, b : B]\!] \to [\![\Gamma]\!].$$

That is, the sigma type over a type is left adjoint to the weakening that determines that type; this left adjoint, as we proved in Proposition 4.25, is given by postcomposition with π_A. Thus, categorically, the type $\sum_{a:A} B(a)$ is given in the empty context by the composition of the projections that give rise to the type A and to the type B in the context of A.

$$[\![x : A, y : B]\!] \xrightarrow{\pi_B} [\![x : A]\!] \xrightarrow{\pi_A} 1$$

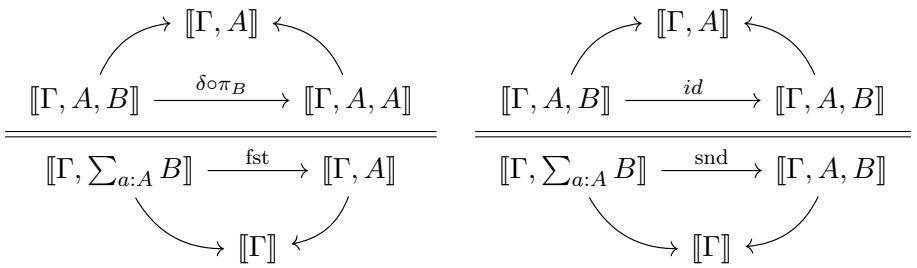
Thus, elements of this type can be built categorically with an element of $a : A$, using a pullback to create the context $[\![B(a)]\!]$ and then providing an element $b : B(a)$.

$$\begin{array}{ccccc} & & \overset{\langle a,b\rangle}{\frown} & & \\ [\![\Gamma]\!] & \xrightarrow{b} & [\![\Gamma, y : B(a)]\!] & \longrightarrow & [\![\Gamma, x : A, y : B]\!] \\ & \searrow^{=} & \downarrow & & \downarrow{\scriptstyle \pi_B} \\ & & [\![\Gamma]\!] & \xrightarrow{a} & [\![\Gamma, x : A]\!] \\ & & & \searrow^{=} & \downarrow{\scriptstyle \pi_A} \\ & & & & [\![\Gamma]\!] \end{array} \qquad \pi_\Sigma$$

This can be rewritten as the following introduction rule.

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B[a/x]}{\Gamma \vdash \langle a, b\rangle : \sum_{x:A} B}$$

The adjunction in the slice category can be then particularized in the following two cases, taking $\delta \colon [\![\Gamma, A]\!] \to [\![\Gamma, A, A]\!]$ to be the substitution that simply duplicates the A.



These two equalities represent two elimination rules.

$$\frac{\Gamma \vdash m : \sum_{x:A} C}{\Gamma \vdash \mathtt{fst}(m) : A} \qquad \frac{\Gamma \vdash m : \sum_{x:A} C}{\Gamma \vdash \mathtt{snd}(m) : C[\mathtt{fst}(m)/a]}$$

We have two beta rules $\mathtt{fst}\langle a, b\rangle \equiv a$ and $\mathtt{snd}\langle a, b\rangle \equiv b$, and a uniqueness rule $m \equiv \langle \mathtt{fst}(m), \mathtt{snd}(m)\rangle$.

4.4.5 Dependent functions

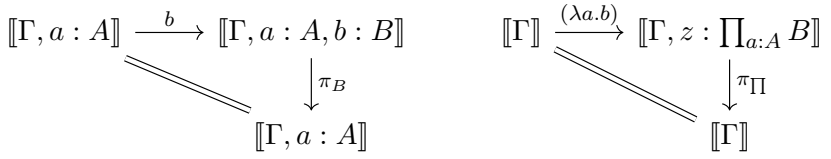
Dependent function types, or **Π-types**, can be seen as a generalized version of function types. Given a family of types parametrized by another type, $\{B(a)\}_{a:A}$, the elements of $\prod_{x:A} B(x)$ are functions with the type A as domain and a changing codomain $B(x)$, depending on the specific element x to which the function is applied. This type is often written also as $\Pi(x : A), B(x)$ to resemble the universal quantifier; under the *propositions as types* interpretation, it would correspond to the proof that a proposition B holds for any $x : A$, that is, $\forall(x : A), B(x)$.

In a locally closed cartesian category, given a type A in a context Γ, as $\pi_A \colon [\![\Gamma, a : A]\!] \to [\![\Gamma]\!]$; and B a type depending the context $\Gamma, a : A$, as $\pi_B \colon [\![\Gamma, a : A, b : B]\!] \to [\![\Gamma, a : A]\!]$, the type $\prod_{a:A} B$ is given by the object

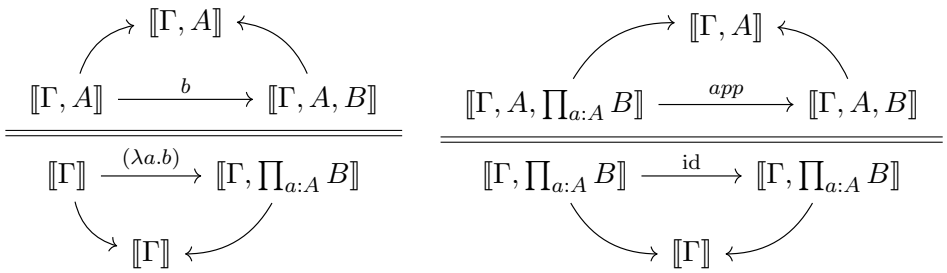
$$\Pi_{\pi_A} \pi_B \colon [\![\Gamma, a : A, b : B]\!] \to [\![\Gamma]\!].$$

That is, the dependent function type over a type is right adjoint to the weakening that determines that type.

Thus, categorically, the type $\prod_{a:A} B(a)$ is given in the empty context as the adjoint $\pi_A^* \dashv \Pi_{\pi_A}$ of the morphism representing the type B. Elements of this type can be built applying the adjunction on the diagram of any term of type B that assumes $a : A$ in the context.



That is, we have the following adjunction and counit in the slice category



which can be rewritten as introduction and elimination rules.

$$\frac{\Gamma, a : A \vdash b : B}{\Gamma \vdash (\lambda a.b) : \prod_{a:A} B} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash f : \prod_{a:A} B}{\Gamma \vdash f\ a : B(a)}$$

We have the equalities $(\lambda a.b)\ a' \equiv b[a'/a]$ and $(\lambda a.f\ a) \equiv f$ as beta and eta rules.

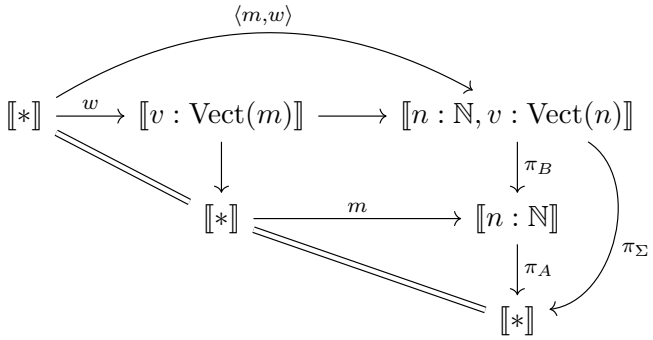
4.5 Working in locally cartesian closed categories

4.5.1 Examples of dependent types

Example 4.28 (Vectors). A common example of dependent types are vectors of elements of a fixed type A varying in length. They can be described as a family of types $\mathrm{Vect}(n)$ depending on a parameter $n \in \mathbb{N}$. For example, given $a_0, a_1, a_2 : A$, we can construct the element $(a_0, a_1, a_2) : \mathrm{Vect}(3)$. We will study what it means to give an element of the types

$$\sum_{n:\mathbb{N}} \mathrm{Vect}(n) \quad \text{and} \quad \prod_{n:\mathbb{N}} \mathrm{Vect}(n).$$

In the first case, we can build an element of $\sum_{n:\mathbb{N}} \mathrm{Vect}(n)$ following the characterization of the adjunction.



That is, we have to provide a morphism from the empty context to the naturals $m : [\![*]\!] \to [\![n : \mathbb{N}]\!]$, or, in other words, an element of $\mathbb{N}$. Computing the pullback of the dependent type of vector over this element gives us the context $[\![v : \mathrm{Vect}(m)]\!]$ of vectors of length m. Now, we only have to provide a morphism $w\colon [\![*]\!] \to [\![v : \mathrm{Vect}(m)]\!]$, or, in other words, a vector of m elements.

In conclusion, elements of $\sum_{n:\mathbb{N}} \mathrm{Vect}(n)$ are of the form $\langle m, w\rangle$, where $m : \mathbb{N}$ and $w : \mathrm{Vect}(m)$. An example would be $\langle 2, (a_0, a_1)\rangle : \sum_{n:\mathbb{N}} \mathrm{Vect}(n)$.

In the second case, we can build an element of $\prod_{n:\mathbb{N}} \mathrm{Vect}(n)$ following the adjunction.

$$\begin{array}{ccc} [\![n : \mathbb{N}]\!] & \xrightarrow{t} & [\![n : \mathbb{N}, v : \mathrm{Vect}(n)]\!] \\ & \searrow & \downarrow{\scriptstyle \pi_{\mathrm{Vect}}} \\ & & [\![n : \mathbb{N}]\!] \end{array} \qquad \begin{array}{ccc} [\![*]\!] & \xrightarrow{(\lambda n.t)} & [\![v : \prod_{n:\mathbb{N}} \mathrm{Vect}(m)]\!] \\ & \searrow & \downarrow{\scriptstyle \pi_{\Pi}} \\ & & [\![*]\!] \end{array}$$

That is, we have to provide a morphism $[\![n : \mathbb{N}]\!] \to [\![n : \mathbb{N}, v : \mathrm{Vect}(n)]\!]$ making the first diagram commute. This is to build a term $t : \mathrm{Vect}(n)$ assuming a given $n : \mathbb{N}$ in the context; in other words, a function sending each n to a vector of n elements. Once it is built, the adjunction gives us a term $(\lambda n.t) : \prod_{n:\mathbb{N}} \mathrm{Vect}(n)$ in the empty context.

In conclusion, elements of $\prod_{n:\mathbb{N}} \mathrm{Vect}(n)$ are functions sending each natural to a vector of that length. An example would be a function $(\lambda m.(a_0, \overset{m}{\ldots}, a_0))$ sending each m to a vector of repeated a_0's.

Example 4.29 (The theorem of choice). A proposition P could be given as a family of types parametrized over another type A, where $P(a)$ is inhabited if and only if the proposition holds for $a : A$.

A proof of the existential quantifier $\sum_{a:A} P(a)$ would be given by a pair $\langle a, p\rangle$, where $a : A$ would be a fixed element and $p : P(a)$ would be the proof that the proposition holds for a. A proof of the universal quantifier $\prod_{a:A} P(a)$ would be a function sending each $a : A$ to a proof $p : P(a)$.

As an example, suppose a relation R, a family of types parametrized over two types A, B. We will prove the following theorem

$$\left(\prod_{(a:A)} \sum_{(b:B)} R(a, b)\right) \to \left(\sum_{(g:A\to B)} \prod_{(a:A)} R(a, g(a))\right).$$

Proof. In order to prove the implication, we assume that we have a term $f : \prod_{a:A} \sum_{b:B} R(a, b)$, and construct a term of type $\sum_{g:A\to B} \prod_{a:A} R(a, g(a))$. The first step is to provide a function of type $A \to B$, and $g :\equiv \lambda a.\mathtt{fst}(f(a))$ is exactly of this type. The second step is to provide a term $\prod_{a:A} R(a, \mathtt{fst}(f(a)))$, but now the function $\lambda a.\mathtt{snd}(f(a))$ is exactly of this type. □

Surprisingly, the theorem we have just proved reads clasically as

> "If for all $a : A$ there exists a $b : B$ such that $R(a, b)$, then there exists a function $g : A \to B$ such that for each $a : A$, $R(a, g(a))$."

and this is a form of the **axiom of choice**. Have we just proved the axiom of choice? The key insight here is that Σ must not be read as the classical "there exists", but as a constructivist existential quantifier that demands not only to merely prove that something exists but to explicitly construct it. A more accurate read would be

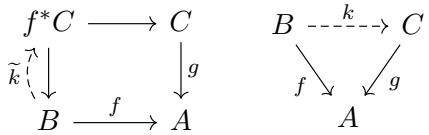
> "If for all $a : A$ we have a rule to explicitly construct a $b : B$ such that $R(a, b)$, then we can use that rule to define a function $g : A \to B$ such that for each $a : A$, $R(a, g(a))$."

and this trivial-sounding theorem is known as the **theorem of choice**. The moral of this example is that, when we work in type theory, we are working inside constructivist mathematics and the existential quantifier has a fundamentally different meaning. Later, we will see a technique that will allow us to recreate the classical existential quantifier.

4.5.2 Equality types

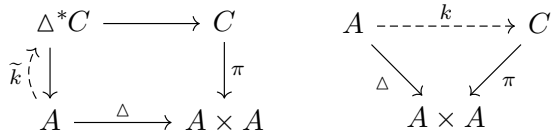
Equality in our theory comes from an adjunction, as was first proposed in [Law70]. The equality type between elements of type A will be represented by the diagonal morphism $\triangle\colon A \to A \times A$ as an object of $\mathcal{C}/(A \times A)$. It is thus a type parametrized over two elements $x, y : A$, written as $(x = y)$. An element of an equality type, $p : x = y$ must be read as a proof that x and y are equal.

In general we have that, for any two objects in a slice category, $f\colon B \to A$ and $g\colon C \to A$, morphisms $f \to g$ correspond naturally to sections of the pullback $f^*(g)\colon f^*C \to B$; as in the following diagram.



Given any k in the diagram above, we can construct $\widetilde{k}$ using the universal property of the pullback; conversely, the fact that $\widetilde{k}$ is a section gives us a k by composition with $f^*C \to C$.

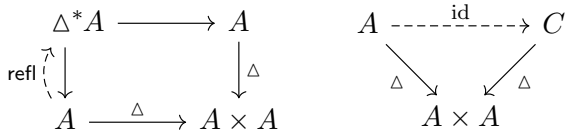
In particular, let $\pi_C\colon C \to A \times A$ or be a family of types $C(x, y)$ parametrized by two elements $x, y : A$. We have that any morphism from the equality type to C corresponds to a section to $\triangle^*C$, which is, by substitution, the family of types $C(x, x)$.



A section $\widetilde{k}$ of this form is precisely a term $x : A \vdash c : C(x, x)$, while a map k is a term $x : A, y : A, p : x = y \vdash c : C(x, y)$. Thus, we have the following elimination rule for equality types, called **J-eliminator** in type theory literature. See [Shu17] for details.

$$\frac{\begin{array}{cc}\Gamma \vdash a : A & \Gamma \vdash b : A \\ \Gamma, x : A \vdash c : C(x, x) & \Gamma \vdash p : a = b\end{array}}{\Gamma \vdash \mathsf{J}_C(c, p) : C(a, b)}$$

The rule informally says that, if we want to prove some property $C(a, b)$ for each $a, b : A$, and we have $a = b$, we only need to prove $C(x, x)$ for each $x : A$. Moreover, if we consider the unit of the adjunction, as shown in the following diagram,



we have a section $\mathsf{refl}\colon A \to \triangle^*A$ that expresses reflexivity, $x = x$ for each $x : A$, and corresponds to the following introduction rule.

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{refl}_a : a = a}$$

Introduction and elimination rules are related by $\mathsf{J}_C(c, \mathsf{refl}_a) \equiv c[x/a]$; that is, the J-eliminator can return c in the reflexivity case.

We can still generalize the rule to allow C to be a family of types also parametrized over p, of the form $C(x, y, p)$.

$$\frac{\begin{array}{cc}\Gamma \vdash a : A & \Gamma \vdash b : A \\ \Gamma, x : A \vdash c : C(x, x, \mathsf{refl}_x) & \Gamma \vdash p : a = b\end{array}}{\Gamma \vdash \mathsf{J}_C(c, p) : C(a, b, p)}$$

The corresponding computation rule would be $\mathsf{J}_C(c, \mathsf{refl}) \equiv c[a/x]$.

4.5.3 Subobject classifier and propositions

A **subobject classifier** is an object Ω with a monomorphism $\mathrm{true}\colon 1 \to \Omega$ such that, for every monomorphism $m : S \to X$, there exists a unique χ such that

$$\begin{array}{ccc} S & \longrightarrow & 1 \\ {\scriptstyle m}\downarrow & & \downarrow{\scriptstyle \mathrm{true}} \\ X & \xrightarrow{\chi} & \Omega \end{array}$$

is a pullback square.

Now, if we have a type given by a monomorphism, $[\![\Gamma, x : P]\!] \to [\![\Gamma]\!]$, by the defining property of the subobject classifier, we have a unique characteristic morphism $P\colon [\![\Gamma]\!] \to [\![\Omega]\!]$, which can be read as $\Gamma \vdash P\colon \Omega$, meaning that Ω is a type whose elements are themselves types (see [Shu17]).

$$\begin{array}{ccc} [\![\Gamma, x : P]\!] & \longrightarrow & 1 \\ \downarrow & & \downarrow{\scriptstyle \mathrm{true}} \\ [\![\Gamma]\!] & \xrightarrow{\chi_P} & [\![\Omega]\!] \end{array}$$

A type P is determined by a monomorphism if, equivalently, any two of its elements are equal. Thus, the elements of Ω are the types with at most one element; these types are usually called **propositions** in type theory. This can be expressed by the following rule.

$$\frac{\Gamma \vdash P : \Omega \qquad \Gamma \vdash a : P \qquad \Gamma \vdash b : P}{\Gamma \vdash \mathrm{isProp}_P(a, b) : a = b}$$

Any two proofs of a proposition (in the sense of type theory) must be equal, and thus, propositions allow us to reintroduce the notion of proof irrelevance.

4.5.4 Propositional truncation

Propositions are types, but not all types are propositions; a type A may have multiple distinct elements, witnessing different proofs of the same fact. We could, however, *truncate* a type into a proposition $\|A\|$ by postulating that any two of its proofs $p, q : \|A\|$ should be equal, $p = q$. In this case, to provide an element of A would mean to explicitly construct a proof of A;

whereas to provide an element of $\|A\|$ would mean to witness that A can be proved, without constructing any proof.

For instance, there are four distinct elements of

$$\sum_{(n,m):\mathbb{N}\times\mathbb{N}} (n+m=3),$$

each one of them providing an ordered pair of naturals that add up to 3. In contrast, there is a unique element of

$$\left\| \sum_{(n,m):\mathbb{N}\times\mathbb{N}} (n+m=3) \right\|,$$

that simply witnesses the existence of some pair of naturals that add up to 3, without explicitly pointing to it. In this sense, the truncated version resembles more the existential quantifier of classical logic; while the untrucated version demands the explicit construction of an example.

Example 4.30. As a second example, we can reformulate a nontrivial version of the axiom of choice we discussed previously in Example 4.29. Note that

$$\left(\prod_{(a:A)} \left\| \sum_{(b:B)} R(a,b) \right\| \right) \to \left\| \sum_{(g:A\to B)} \prod_{(a:A)} R(a,g(a)) \right\|,$$

now represents the fact that we want to obtain evidence of the existence of a function only knowing that for each a there exists an element b related to it, but (crucially) without knowing which b is related to a. This new version of the Axiom of Choice is indeed independent of our theory (see Chapter 3 of [Uni13]).

How should we represent truncations inside our theory? It can be proved (see [AB04]) that propositional truncation is the left adjoint to the inclusion of propositions into general types; and, if we assume the existence of this adjoint into the category we are working into, we can use propositional truncations. That is, if P is a proposition and A is an arbitrary type, we have the following adjunction

$$\frac{A \longrightarrow P}{\|A\| \longrightarrow P}$$

which in practical terms means that, if we want to prove $\|A\| \to P$, where P is a proposition, it suffices to prove $A \to P$, that is, to assume a particular proof for A. Note that this corresponds to usual mathematical practice, where having a proof of existence can be used to assume that we have explicitly constructed the element and to prove a different proposition using it, with the condition that we cannot refer later to the explicit element we used during the proof.

4.6 Topoi

4.6.1 Motivation

Topoi (singular *topos*) are category-theoretic models of constructive mathematics; we can reason in its internal logic and rebuild large parts of mathematics inside their structure. Each

topos is thus an universe of mathematics with different axioms and interpretations [Bau17]; for example,

- inside the Hayland's **realizability topos**, every function is computable and we can study Kleene's realizability theory (see [VO08]);
- the **Dubuc topos** provides a non paradoxical formalization of notion of "infinitesimal" used by Newton and Leibniz, and we can study synthetic differential geometry inside it (see [Dub89]);
- inside the Johnstone's **topological topos**, we can reason with topological spaces and continuous functions between them (see [Joh79]), and under certain hypothesis, we can assume that all functions we can build are automatically continuous (see [HEX15]).

Topoi are defined as locally cartesian closed categories with a subobject classifier in which every finite limit exists. Usually, we will be interested in **W-topoi**, or topoi with a natural numbers object (in the sense of Example 3.46) [Lei10].

The study of any of these theories is beyond the scope of this text. However, as we have been relating type theory to the internal language of locally closed cartesian categories with enough structure; we know that type theory (depending on what constructions we assume) will have models in categories like these.

4.6.2 An Elementary Theory of the Category of Sets

Lawvere's Elementary Theory of the Category of Sets [Law64] provides a foundation for mathematics based on category theory. It describes the category Set in an abstract setting using eight axioms; and the atomic, undefined notions of the theory are not memberships and sets, but morphisms and composition. In the original article some examples on how to do set theory inside the category are shown.

Using the notation we have developed so far, Lawvere's axioms can be reduced to the following definition. A model of the Elementary Theory of the Category of Sets is

- a *topos*, that is, a locally closed cartesian category with all finite limits and a subobject classifier,
- which is ***well-pointed***, meaning that any two morphisms $f, g \colon A \to B$ are equal if and only if $f \circ a = g \circ a$ for each $a \colon 1 \to A$; morphisms from the terminal object are called *global elements*, and this property can be thought as function extensionality;
- which has a natural numbers object, in the sense of Example 3.46;
- which satisfies the **Axiom of Choice**, meaning that point-surjective morphisms have a section; in terms of our previous Example 4.30, we could translate this internally as

$$\left(\prod_{(a:A)} \left\| \sum_{(b:B)} f(b) = a \right\| \right) \to \left\| \sum_{(g:A\to B)} \prod_{(a:A)} f(g(a)) = a \right\|,$$

  for any $f \colon B \to A$.

Note that the category Set, under the usual axioms, is a model of this theory. This can be seen, then, as an abstraction of set theory. As we will see later, the Axiom of Choice implies the Law of Excluded Middle, so we have finally recovered a classical foundation of mathematics from category theory.

Chapter 5

Type theory

5.1 Martin-Löf type theory

> And it soon became clear that the only long-term solution was somehow to make it possible for me to use computers to verify my abstract, logical, and mathematical constructions. When I first started to explore the possibility, computer proof verification was almost a forbidden subject among mathematicians. The primary challenge that needed to be addressed was that the foundations of mathematics were unprepared for the requirements of the task.
>
> – **Vladimir Voevodsky**, [voe].

In this chapter, we will exclusively work internally in dependent type theory and use it as a constructive foundation of mathematics. This will have the added benefit that every formal proof in the system will be a closed lambda term and checking that a proof is correct will amount to typechecking the term. Explicitly, the type theory we have been describing in the previous sections corresponds to **Martin-Löf type theory** [NPS90].

5.1.1 Programming in Martin-Löf type theory

Martin-Löf type theory and the internal language we have been describing so far can also be regarded as a programming language in which is possible to formally specify and prove theorems about the code itself. Formal proofs in this theory can be written in any language with a powerful enough type system; examples of these include

- **Agda**, a programming language that implements a variant of Martin-Löf type theory;
- **Coq** [04] was developed in 1984 in INRIA; it implements Calculus of Constructions and was used, for example, to check a proof of the Four Color Theorem;
- **Idris**, a programming language implementing dependent types and using a slightly modified version of intensional equality types;
- **NuPRL** [CAB+86], a proof assistant implementing Martin-Löf *extensional* Type Theory, which is different from the intensional theory in how it defines equality types;
- **Cubical** [CCHM16] and **RedPRL** [SGR+16] provide experimental implementations of Cubical Type Theory, a different variant of type theory.

In this text, we will use Agda to write mathematics, taking ideas from [McB17]. We will check proofs in Martin-Löf type theory using its type system.

5.1.2 Translation between categories and types

We can translate the categorical structure inside Agda as follows. Dependent products exist naturally as dependent functions of the language.

$$\frac{\Gamma, a : A \vdash b : B}{\Gamma \vdash (\lambda a.b) : \prod_{a:A} B}$$

```
f : (a : A) → B a
f = λ a → b
```

Dependent sums must be explicitly specified in the form of *records*: data structures in which the type of every element can depend on the previous ones.

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B[a/x]}{\Gamma \vdash \langle a, b \rangle : \sum_{x:A} B}$$

```
record Σ (S : Set) (T : S → Set) : Set
  where
  constructor _,_
  field
    fst : S
    snd : T fst
```

Naturals, and initial algebras in general, can be defined as inductive types with the *data* keyword. Functions over these types can be constructed using their universal property, as we did in Example 3.46.

$$\begin{array}{ccc} 1 + \mathbb{N} & \longrightarrow & 1 + \hom(\mathbb{N}, \mathbb{N}) \\ {\scriptstyle \langle 0, \mathrm{succ} \rangle}\downarrow & & \downarrow{\scriptstyle \langle \mathrm{id}, \mathrm{succ} \circ - \rangle} \\ \mathbb{N} & \xrightarrow{+} & \hom(\mathbb{N}, \mathbb{N}) \end{array}$$

```
data ℕ : Set where
  zero : ℕ
  succ : (n : ℕ) → ℕ

_+_ : ℕ → (ℕ → ℕ)
zero + m = m
succ n + m = succ (n + m)
```

Equality types are a particular case of an inductive family of types. The induction principle over equalities is the *J-eliminator* we described in Section 4.5.2.

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{refl}_a : a = a}$$

```
data _≡_ {A : Set} : A → A → Set where
  refl : {a : A} → a ≡ a
```

Using all this machinery, we can prove facts about equality by induction and prove facts about the natural numbers, as in the following example, where we prove that $a = b$ implies $f(a) = f(b)$ and then use this fact to prove that $n + 0 = n$ for any $n : \mathbb{N}$ by induction.

```
ap : {A B : Set} (f : A → B) {a b : A} → a ≡ b → f a ≡ f b
ap f refl = refl
```

```
+rzero : (n : ℕ) → n + 0 ≡ n
+rzero 0 = refl
+rzero (succ n) = ap succ (+rzero n)
```

Finally, we can define *propositions* as those types where any two elements are equal and postulate that truncations are propositions. The induction principle for truncated types represents the adjunction we described in Section 4.5.3.

$$\frac{\Gamma \vdash P : \Omega \qquad \Gamma \vdash a : P \qquad \Gamma \vdash b : P}{\Gamma \vdash \mathrm{isProp}_P(a, b) : a = b}$$

```
isProp : Set → Set
isProp A = ((x y : A) → x ≡ y)
postulate trunc : {A : Set} → isProp ∥ A ∥
```

$$\frac{A \longrightarrow P}{\|A\| \longrightarrow P}$$

```
trunc-rec : {A : Set} {P : Set} → isProp P
  → (A → P)
  → ∥ A ∥ → P
trunc-rec _ f  x  = f x
```

5.1.3 Excluded middle and constructivism

Using categories, we have strengthen the propositional logic we described in Section 1.3.2 to a fully-fledged higher order logic in which we can construct mathematical proofs. However, during all this process, we have not accepted the Law of Excluded Middle; we do not assume that $P \vee \neg P$ for an arbitrary proposition P. Note, however, that we do not negate it, neither: that would cause a contradiction, as we actually proved $\neg\neg(P \vee \neg P)$ in general in Section 2.6.3. We are *agnostic* regarding it.

Why are we not simply accepting it, as is common practice? The Law of Excluded middle would affect the computational properties of the theory. For instance, whenever we create a natural number in our theory, we expect it to be a numeral of the form $\mathrm{succ}(\mathrm{succ}(\ldots \mathrm{zero} \ldots))$ for an some number of succ applications. Inductive definitions can only compute with natural numbers when they are on this form. However, if we were to postulate the Law of Excluded Middle, LEM: $P \vee \neg P$, we should accept the existence of numbers such as

$$\text{if LEM(RiemmanHypothesis) then 0 else 1,}$$

but we would not be able to compute them into a numeral. We say that a type system has the ***canonicity property*** if every inhabitant of the $\mathbb{N}$ type is a numeral.

In any case, classical mathematics is a particular case of constructive mathematics. We can choose to work inside classical mathematics, although we would lose the computational content. When we choose not to, we are working in constructive mathematics with a programming language as it was first proposed by Errett Bishop [Bis67].

5.1.4 Extensionality and Diaconescu's theorem

If we want to recover classical mathematics, we should try to model the axioms of the Elementary Theory of Sets into our locally closed cartesian categories. We already have natural numbers in Martin-Löf type theory, and *well-pointedness*, for instance, is called *function extensionality* in type theory literature.

Definition 5.1 (Function extensionality)**. Function extensionality** states that, if any two functions $f, g\colon A \to B$ have the same images under the same arguments, they are, in fact, equal. That is,

$$\prod_{x:A} (f(x) = g(x)) \to (f = g).$$

We postulate this in Agda as follows.

```
postulate
  wellPointed : {A B : Set} → {f g : A → B}
    → ((x : A) → f x ≡ g x)
    → f ≡ g
```

The last ingredient is the Axiom of Choice. This, like in the case of the Law of Excluded Middle, will make us lose the computational content of the theory. In fact, we can prove that, in general, the Axiom of Choice implies the Law of Excluded Middle. This is the statement of **Diaconescu's theorem**, for which we provide both a natural proof and a computer-verified proof.

Theorem 5.2 (R. Diaconescu, 1975)**.** *The Axiom of Choice implies the Law of Excluded Middle.*

Proof. The proof we present is based in [Alt] and [Bau17]. Given any proposition P, we define $U = \{x \in \{0,1\} \mid (x = 0) \vee P\}$ and $V = \{x \in \{0,1\} \mid (x = 1) \vee P\}$, and we kwow that each one is inhabited. By the axiom of choice, there exists a function $f\colon \{U, V\} \to U \cup V$ such that $f(U) \in U$ and $f(V) \in V$. We decide if $f(U)$ and $f(V)$ are equal to 0 or not (by induction). If $f(U) = 1$ or $f(V) = 0$, we would have that P must be true; and if $f(U) = 0$ and $f(V) = 1$, we would have $\neg P$, for if P were true, then U would be equal to V and thus, $0 = f(U) = f(V) = 1$. □

This proof has been implemented as follows. Note that we have only assumed the Axiom of Choice (and therefore, the Law of Excluded Middle) during this particular section of the text. In the next section, we return to constructive mathematics, working with the real numbers and extracting algorithms from proofs.

```
postulate
  AxiomOfChoice : {A : Set} {B : Set} {R : A → B → Set}
    → ((a : A) → (∃ b ∈ B , (R a b)))
    ----------------------------------------
    → (∃ f ∈ (A → B), ((a : A) → R a (f a)))

LawOfExcludedMiddle : {P : Set} → P ∨ ¬ P
LawOfExcludedMiddle {P} = Ex-elim
  (AxiomOfChoice λ { (Q , q) → Ex-elim q Ex-isProp λ { (u , v) → u ,, v }})
  ∨-isProp λ { (f , α) → byCases f α }
  where
    A : Set
    A = Σ (Bool → Set) (λ Q → Ex Bool (λ b → Q b))
```

```
R : A → Bool → Set
R (P , _) b = P b

U : Bool → Set
U b = (b ≡ true) ∨ P
V : Bool → Set
V b = (b ≡ false) ∨ P
Ua : A
Ua = U , (true ,, rinl refl)
Va : A
Va = V , (false ,, rinl refl)

module lemma (f : A → Bool) where
  eqf : (p : P) → f Ua ≡ f Va
  eqf p = ap f (Σ-eq Ua Va (
    wellPointed λ
      { false → propext ∨-isProp ∨-isProp (λ _ → rinr p) (λ _ → rinr p)
      ; true → propext ∨-isProp ∨-isProp (λ _ → rinr p) (λ _ → rinr p)
      }) (Ex-isProp _ _))

  refute : true ≡ false → P ∨ ¬ P
  refute ()
  byCases : (α : (x : A) → R x (f x)) → P ∨ ¬ P
  byCases α with f Ua ?? | f Va ??
  byCases α | inl x | inr y = rinr λ p → true≠false (inv x · (eqf p · y))
  byCases α | inl x | inl y = ∨-elim (α Va) ∨-isProp
    λ { (inl q) → refute (inv y · q) ; (inr p) → rinl p }
  byCases α | inr x | inl y = ∨-elim (α Ua) ∨-isProp
    λ { (inl q) → refute (inv q · x) ; (inr p) → rinl p }
  byCases α | inr x | inr y = ∨-elim (α Ua) ∨-isProp
    λ { (inl q) → refute (inv q · x) ; (inr p) → rinl p }
open lemma public
```

5.1.5 Dedekind reals

In `Reals.agda`, we provide a formalized construction of the Dedekind positive reals in intensional Martin-Löf type theory, implemented in Agda. This implementation constructs reals as Dedekind cuts over the **positive dyadic rationals** D; that is, over the rationals of the form $a/2^b$ for some $a, b : \mathbb{N}$. We also provide a library with all the necessary lemmas proving that our constructions are well-defined.

Natural numbers are constructed as initial algebras (as described in Example 3.46). Dyadic rationals are constructed as pairs of naturals endowed with a normalization property: a fraction $a/2^b$ is *normalized* if a is an odd number or if b is exactly zero.

$$D = \sum_{(a,b):\mathbb{N}\times\mathbb{N}} \|\mathrm{odd}(a) + \mathrm{isZero}(b)\|$$

This technique ensures that each dyadic rational will be uniquely represented by a term of type D. Finally, a real number $r : \mathbb{R}^+$ is constructed as the sum of the following data.

- A **Dedekind cut**, $\mathrm{cut}_r : D \to \Omega$, representing a proposition parametrized over the positive dyadic rationals. Given $q : D$, the proposition $\mathrm{cut}_r(q)$ is true if and only if $r < q$.
- A proof $\| \sum_{q:D} \mathrm{cut}_r(q) \|$ witnessing that the Dedekind cut is **inhabited**, that is, there exists some q such that $r < q$, providing an upper bound on the real number.
- Two functions that witness that the cut is **round**. That is, a dyadic $q : D$ is in the cut if and only if there is a smaller $p : D$ also in the cut. Symbolically,

$$\left(\prod_{q:D} \mathrm{cut}_r(q) \to \left\| \sum_{p:D} (p < q) \times \mathrm{cut}_r(p) \right\|\right) \times \left(\prod_{q:D} \left\| \sum_{p:D} (p < q) \times \mathrm{cut}_r(p) \right\| \to \mathrm{cut}_r(q)\right).$$

The following code shows the core definition of the Agda implementation.

```
record ℝ⁺ : Set where
  constructor real
  field
    cut : F → Set
    isprop : (q : F) → isProp (cut q)

    bound : ∃ q ∈ F , cut q
    round1 : (q : F) → cut q → ∃ p ∈ F , ((p < q ≡ true) × cut p)
    round2 : (q : F) → (∃ p ∈ F , ((p < q ≡ true) × cut p)) → cut q
open ℝ⁺ {{...}} public
```

Note that, under the intuitionistic interpretation, it is not the case in general that $\prod_{q:D} \mathrm{cut}_r(q) + \neg\, \mathrm{cut}_r(q)$ for an arbitrary $r : \mathbb{R}^+$; in fact, numbers that are neither equal nor distinct from zero could exist! as we mentioned earlier, a formalization of infinitesimals is possible using this property, see [Bau13]. However, even if we cannot prove it in general, we can prove $\mathrm{cut}_r(q) + \neg\, \mathrm{cut}_r(q)$ for some particular real numbers. We call these numbers **located**, and, for these numbers, the computational nature of the proof provides an algorithm that produces an infinite stream with the digits of the number.

As a proof of concept, we define square roots, $\sqrt{-} : \mathbb{R}^+ \to \mathbb{R}^+$, as

$$\mathrm{cut}_{\sqrt{r}}(q) = \left\| \sum_{p:D} (\mathrm{cut}_r(p)) \times \left(p < q^2\right) \right\|.$$

We prove they are well-defined using several lemmas about the dyadic numbers and we prove that $\sqrt{2}$ is located. The Agda compiler is then able to produce the first binary digits of $\sqrt{2} = 1.01101010000\ldots$. Note that this construction is thus being formalized in any locally cartesian closed category with enough structure. Explicitly, we use the following Agda code; it calls to already defined lemmas on the natural numbers when necessary and it shows how existential elimination is written inside the language to prove the necessary properties of the Dedekind cut.

```
sqrt : ℝ⁺ → ℝ⁺
sqrt a = record
  { cut = λ f → ∃ g ∈ F , (cut {{a}} g × (g < f * f ≡ true))
  ; isprop = λ f → Ex-isProp
  ; bound = Ex-elim (bound {{a}}) Ex-isProp λ { (g , α) → one + g ,, (g ,, (α , F-lemma3 g)) }
  ; round1 = λ f x →
      Ex-elim x Ex-isProp λ { (g , (α , β)) →
      Σ-elim (<sqbetween g (f * f) β) λ { (r , (γ , δ)) →
```

```
    r ,, (<sqless r f δ , (g ,, (α , γ)))
    }}
; round2 = λ f x →
    Ex-elim x Ex-isProp λ { (r , (α , h)) →
    Ex-elim h Ex-isProp λ { (u , (β , p)) →
    u ,, (β , F-lemma4 u f r p α)
    }}
}
```

Dedekind cuts are not the only path we can take in order to define the real numbers in type theory. Reals can be also defined in terms of Cauchy sequences, but the resulting construction is not in general equivalent to ours. Only when we assume excluded middle, both Dedekind and Cauchy reals become equivalent and the classical notion of real number is recovered. A detailed discussion can be found in the last chapters of [Uni13].

5.2 Homotopy type theory

5.2.1 Homotopy type theory I: Equality

We have already discussed how, given any $x, y : A$, we can interpret the equality type $(x = y)$, whose terms are witnesses of the equality. For each $x : A$, there is a reflexivity element, $\mathsf{refl} : x = x$; and the J-eliminator is interpreted as the following induction principle over the type.

$$\prod_{(C:\prod_{(x,y:A)}(x=y)\to\mathcal{U})} \left(\left(\prod_{a:A} C(a, a, \mathsf{refl}) \right) \to \prod_{(x,y:A)} \prod_{(p:x=y)} C(x, y, p) \right).$$

This induction principle allows us to prove symmetry, transitivity and other properties of equality (we explicitly do so in `Base.agda` and `Equality.agda`). However, it is not posible to prove by path induction that every path is equal to the reflexivity path, that is,

$$\prod_{(x:A)} \prod_{(p:x=x)} (p = \mathsf{refl}),$$

is not derivable from the induction principle. This is called the principle of **uniqueness of identity proofs**, it is equivalent to Streicher's Axiom K [Str93] and it is independent from Martin-Löf type theory.

If we do not assume this axiom, we open the possibility to the existence of multiple different proofs of the same equality. The structure of these proofs, endowed with symmetry and transitivity, could be modeled into a groupoid; and this idea allows us to construct models of type theory where the principle of uniqueness of identity proofs does not hold (see [HS98]). If we also consider equalities between proofs of equality, and equalities between proofs of equalities between equalities, and so on, we would get a weak ω-groupoid structure [VDBG11]. Following the **Grothendieck's Homotopy Hypothesis**, groupoids can be regarded as an homotopical spaces, where equalities are paths and equalities between equalities are homotopies between paths (some work on this hypothesis can be read in [Tam96]).

In any case, as the existence of this non-trivial structure is independent of the theory, we need to introduce new axioms or new types with nontrivial equalities if we want to profit from

this interpretation. The introduction of Voevodsky's Univalence Axiom leads to **Homotopy Type theory**, an extension of Martin-Löf type theory where we can work with this groupoid structure. Under the identification of higher groupoids and homotopical types, the new axiom allows us to reason in some sort of synthetic homotopy theory, where paths and homotopies are primitive notions. For instance, we can define the **fundamental group** (fundamental groupoid, if we also want to consider higher structure) of a type A in a point $a : A$ as the type of loops $\pi_1(A, a) :\equiv a = a$, endowed with reflexivity and transitivity. The **circle** can be defined as the freely generated type with a single element $\mathsf{p} : \mathbb{S}^1$ and a nontrivial equality $\mathsf{loop} : \mathsf{p} = \mathsf{p}$. Because it is freely generated, we can apply symmetry to get a different proof $\mathsf{p} = \mathsf{p}$ which we will call loop^{-1}; moreover, we can transitivity n times to loop for an arbitrary n, to get a new proof of $\mathsf{p} = \mathsf{p}$ which we will call loop^n; these are the elements of its fundamental group $\pi_1(\mathbb{S}^1, \mathsf{p})$.

In this setting, results such as the Van Kampen theorem or the construction of Eilenberg-MacLane spaces have been formalized (see [Uni13]).

5.2.2 Homotopy type theory II: Univalence

We say that there is an equivalence between two types A and B and we write it as $(A \simeq B)$ if there is a *biinvertible* map between them. Explicitly,

$$(A \simeq B) = \sum_{f:A\to B} \left(\left(\sum_{g:B\to A} \prod_{a:A} g(f(a)) = a \right) \times \left(\sum_{g:B\to A} \prod_{b:B} f(g(b)) = b \right) \right).$$

It can be shown that, for any pair of types A and B, there exists a function of type idtoeqv : $(A =_{\mathcal{U}} B) \to (A \simeq B)$ that uses identity functions to construct an equivalence from an equality. The Univalence axiom states that this function is itself an equivalence.

Axiom 5.3 (Univalence)**.** For any pair of types $A, B : \mathcal{U}$, idtoeqv : $(A = B) \to (A \simeq B)$ is an equivalence. In particular, $(A = B) \simeq (A \simeq B)$.

In practical terms, this implies that isomorphic structures can be identified. For example, we could consider the type of the integers $\mathbb{Z}$ with the successor and predecessor functions and create an equivalence $\mathbb{Z} \simeq \mathbb{Z}$ which can be turned into a nontrivial equality $\mathbb{Z} = \mathbb{Z}$ via the Univalence axiom, representing that integers, as a type, are equivalent to themselves after shifting them by the successor function. In the file `FundGroupCircle.agda`, we use this fact to prove, inside type theory, that $\mathbb{Z} = \pi_1(\mathbb{S}^1)$, (following [Uni13] and [LS13]). This result is a proof, inside this synthetic homotopical setting, of the fact that the fundamental group of the circle is $\mathbb{Z}$.

```
-- Winds a loop n times on the circle.
loops : ℤ → Ω S¹ base
loops n = z-act (Ω-st S¹ base) n loop

-- Uses univalence to unwind a path over the integers.
code : S¹ → Type0
code = S¹-ind Type0 ℤ (UnivalenceAxiom zequiv-succ)

-- Creates an equivalence between paths and encodings.
equiv-family : (x : S¹) → (base == x) ≃ code x
```

```
equiv-family x = qinv-≃ (encode x) (decode x , (encode-decode x , decode-encode x))

-- The fundamental group of the circle is the integers.  In this
-- proof, univalence is crucial.  The next lemma will prove that the
-- equivalence in fact preserves the group structure.
fundamental-group-of-the-circle : Ω S¹ base ≃ ℤ
fundamental-group-of-the-circle = equiv-family base

preserves-composition : ∀ n m → loops (n + m) == loops n · loops m
preserves-composition n m = z-act+ (Ω-st S¹ base) n m loop
```

Chapter 6

Conclusions

6.1 Further

- https://arxiv.org/abs/quant-ph/0312174

Chapter 7

Appendices

Acknowlegments

The opportunity of devoting my bachelor's thesis to this fascinating subject has been made possible by Prof. Pedro García-Sánchez and Prof. Manuel Bullejos. They have provided me with corrections and useful guidelines for the text and they have gone beyond their obligation in their efforts to instruct me on how to expose these ideas clearly. Any deviation from this goal must be attributed to my own inexperience.

I would like to thank the LibreIM community in general and Ignacio Cordón, David Charte, Marta Andrés, José Carlos Entrena, Pablo Baeyens, Antonio Checa, Daniel Pozo, and Sofía Almeida in particular, for testing the interpreter and providing useful discussion and questions on the topics of this text. Prof. Luis Merino, Prof. Juan Julián Merelo and Braulio Valdivielso have made possible and participated on the organization of meetings and workshops on these ideas.

Finally, I would like to express my gratitude to Benedikt Ahrens and the organizers of the School and Workshop on Univalent Mathematics at the University of Birmingham, whose effort has given me the opportunity to learn the fundaments of type theory and the Univalent Foundations program.

Bibliography

[AB04] Steve Awodey and Andrej Bauer. Propositions as [types]. *Journal of logic and computation*, 14(4):447–471, 2004.

[AB17] Steve Awodey and Andrej Bauer. Lecture notes: Introduction to categorical logic, 2017.

[Alt] Thorsten Altenkirch. Introduction to homotopy type theory. Lecture notes for a course at EWSCS 2017.

[Awo10] Steve Awodey. *Category theory*. Oxford University Press, 2010.

[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.

[Bau13] Andrej Bauer. Intuitionistic mathematics and realizability in the physical world. In *A Computable Universe: Understanding and Exploring Nature as Computation*, pages 143–157. World Scientific, 2013.

[Bau17] Andrej Bauer. Five stages of accepting constructive mathematics. *Bulletin of the American Mathematical Society*, 54(3):481–498, 2017.

[Bis67] Errett Bishop. A general language, 1967.

[CAB+86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[CCHM16] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. *CoRR*, abs/1611.02108, 2016.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic, Volume I*. North-Holland, 1958. Second printing 1968.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and computation*, 76(2-3):95–120, 1988.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[Cro75] J. N. Crossley. *Reminiscences of logicians*, pages 1–62. Springer Berlin Heidelberg, Berlin, Heidelberg, 1975.

[Cur34] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):584–590, 1934.

[Cur46] Haskell B. Curry. The paradox of Kleene and Rosser. *Journal of Symbolic Logic*, 11(4):136–137, 1946.

[dB72] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381 – 392, 1972.

[DLM08] Ugo Dal Lago and Simone Martini. The weak lambda calculus as a reasonable machine. *Theoretical Computer Science*, 398(1-3):32–50, 2008.

[Dub89] Eduardo J Dubuc. Integración de campos vectoriales y geometría diferencial sintética. *Revista de la Unión Matemática Argentina*, 35:151–162, 1989.

[EM42] Samuel Eilenberg and Saunders MacLane. Group extensions and homology. *Annals of Mathematics*, 43(4):757–831, 1942.

[EM45] Samuel Eilenberg and Saunders MacLane. General theory of natural equivalences. *Transactions of the American Mathematical Society*, 58:231–294, 1945.

[Geu93] Jan Herman Geuvers. *Logics and type systems*. Universiteitsdrukkerij Nijmegen, 1993.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1989.

[HEX15] Martín Hötzel Escardó and Chuangjie Xu. The inconsistency of a Brouwerian continuity principle with the Curry–Howard interpretation. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 38. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[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.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM (JACM)*, 40(1):143–184, 1993.

[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.

[HS98] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. *Twenty-five years of constructive type theory (Venice, 1995)*, 36:83–111, 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.

[Joh79] Peter T Johnstone. On a topological topos. *Proceedings of the London mathematical society*, 3(2):237–271, 1979.

[Jup] Jupyter Development Team. Jupyter Notebooks. A publishing format for reproducible computational workflows.

[Kam01] Fairouz Kamareddine. Reviewing the classical and the de bruijn notation for lambda-calculus and pure type systems. *Logic and Computation*, 11:11–3, 2001.

[Kas00] Ryo Kashima. A proof of the standardization theorem in lambda-calculus. page 6, 09 2000.

[KR35] S. C. Kleene and J. B. Rosser. The inconsistency of certain formal logics. *Annals of Mathematics*, 36(3):630–636, 1935.

[Kun11] K. Kunen. *Set Theory*. Studies in logic. College Publications, 2011.

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

[Lawa] F. William Lawvere. Diagonal arguments and Cartesian Closed Categories.

[Lawb] F. William Lawvere. Use of Logical Operators in mathematics.

[Law64] F William Lawvere. An elementary theory of the category of sets. *Proceedings of the national academy of sciences*, 52(6):1506–1511, 1964.

[Law70] F. William Lawvere. Equality in hyperdoctrines and comprehension schema as an adjoint functor. *Applications of Categorical Algebra*, 17:1–14, 1970.

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

[Lei10] Tom Leinster. An informal introduction to topos theory. *arXiv preprint arXiv:1012.5647*, 2010.

[LS09] F. William Lawvere and Stephen H. Schanuel. *Conceptual Mathematics: A First Introduction to Categories*. Cambridge University Press, New York, NY, USA, 2nd edition, 2009.

[LS13] Daniel R Licata and Michael Shulman. Calculating the fundamental group of the circle in homotopy type theory. In *Logic in Computer Science (LICS), 2013 28th Annual IEEE/ACM Symposium on*, pages 223–232. IEEE, 2013.

[04] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[McB17] Conor McBride. Cs410 advanced functional programming. https://github.com/pigworker/CS410-17, 2017.

[McC91] William W. McCune. Single axioms for groups and abelian groups with various operations. In *Preprint MCS-P270-1091, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL*, 1991.

[ML75] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Studies in Logic and the Foundations of Mathematics*, volume 80, pages 73–118. Elsevier, 1975.

[MM94] I. Moerdijk and S. MacLane. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Universitext. Springer New York, 1994.

[MP00] Ieke Moerdijk and Erik Palmgren. Wellfounded trees in categories. *Annals of Pure and Applied Logic*, 104(1-3):189–218, 2000.

[New17] Clive Newstead. Locally cartesian closed categories. 2017.

[nLa18] nLab authors. HomePage. http://ncatlab.org/nlab/show/HomePage, May 2018. Revision 262.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M Smith. *Programming in Martin-Löf's type theory*, volume 200. Oxford University Press Oxford, 1990.

[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.

[SGR+16] Jonathan Sterling, Danny Gratzer, Vincent Rahli, Darin Morrison, Eugene Akentyev, and Ayberk Tosun. Redprl–the people's refinement logic, 2016.

[Shu17] Michael Shulman. Homotopy type theory: the logic of space. *arXiv preprint arXiv:1703.03007*, 2017.

[Str93] Thomas Streicher. Investigations into intensional type theory. *Habilitiation Thesis, Ludwig Maximilian Universität*, 1993.

[Tai67] W. W. Tait. Intensional interpretations of functionals of finite type i. *The Journal of Symbolic Logic*, 32(2):198–212, 1967.

[Tam96] Z. Tamsamani. Equivalence de la théorie homotopique des n-groupoïdes et celle des espaces topologiques n-tronqués. In *eprint arXiv:alg-geom/9607010*, July 1996.

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

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[VDBG11] Benno Van Den Berg and Richard Garner. Types are weak ω-groupoids. *Proceedings of the London Mathematical Society*, 102(2):370–394, 2011.

[VO08] Jaap Van Oosten. *Realizability: an introduction to its categorical side*, volume 152. Elsevier, 2008.

[voe] The origins and motivations of Univalent Foundations. https://www.ias.edu/ideas/2014/voevodsky-origins. Accessed: 2018-04-17.

[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.

[Yan03] Noson S. Yanofsky. A universal approach to Self-Referential Paradoxes, Incompleteness and Fixed points. pages 15–17, 2003.