CATEGORY THEORY AND LAMBDA CALCULUS

MARIO ROMÁN

UNIVERSIDAD
DE GRANADA

Trabajo Fin de Grado

Doble Grado en Ingeniería Informática y Matemáticas

Tutores
Jesús García Miranda
Pedro A. García-Sánchez

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

Granada, a junio de 2018

CONTENTS

ABSTRACT

This is the abstract.

Part I

CATEGORY THEORY

1

CATEGORIES

1.1 DEFINITION OF CATEGORY

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

Definition 1 (Category)**.** A **category** $\mathcal{C}$, as defined in [Lan78], is given by

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

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

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

Given two morphisms $f \colon A \to B$ and $g \colon B \to C$ there exists a **composition morphism**, written as $g \circ f \colon A \to C$. Morphism composition is a binary associative operation with identity elements $\mathrm{id}_A \colon A \to A$, that is

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

Definition 2 (Hom-sets)**.** The **hom-set** of two objects A, B on a category is the collection of morphisms between them. It is written as $\mathrm{hom}(A, B)$. The set of **endomorphisms** of an object A is the hom-set $\mathrm{end}(A) = \mathrm{hom}(A, A)$.

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

1 : We use the term *collection* to denote some unspecified formal notion of compilation of "things" that could be given by sets or proper classes. We will want to define categories whose objects are all the possible sets and we will need the objects to form a proper class in order to avoid inconsistent results such as the Russell's paradox.

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

Definition 3 (Small and locally small categories)**.** A category is said to be **small** if the collections $\mathcal{C}_0, \mathcal{C}_1$ of objects and morphisms are both sets (instead of proper classes). It is said to be **locally small** if every hom-set is actually a set.

1.2 MORPHISMS

Definition 4 (Isomorphisms)**.** A morphism $f : A \to B$ is an **isomorphism** if an inverse morphism $f^{-1} : B \to A$ such that

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

exists.

We call **automorphisms** to the endomorphisms which are also isomorphisms.

Proposition 1 (Unicity of inverses)**.** *If the inverse of a morphism exists, it is unique. In fact, if a morphism has a left-side inverse and a right-side inverse, they are an inverse, and they are equal.*

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

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

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

Definition 5. Two objects are **isomorphic** if an isomorphism between them exists. We write $A \cong B$ when A and B are isomorphic.

Proposition 2 (Isomorphy is an equivalence relation)**.** *The relation of being isomorphic is an equivalence relation. In particular,*

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

are all isomorphisms.

Definition 6 (Monomorphisms and epimorphisms)**.** A **monomorphism** is a left-cancellable morphism, that is, $f : A \to B$ is a monomorphism if, for every $g, h : B \to A$,

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

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

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

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

Remark 1. A morphism can be a bimorphism without being an isomorphism.

Definition 7 (Retractions and sections)**.** A **retraction** is a left inverse, that is, a morphism which has a right inverse; conversely, a **section** is a right inverse, a morphism which has a left inverse.

By virtue of Proposition 1, a morphism which is both a retraction and a section is an isomorphism.

1.3 TERMINAL OBJECTS, PRODUCTS AND COPRODUCTS

Definition 8 (Initial object)**.** An object I is an **initial object** if every object is the domain of exactly one morphism to it. That is, for every object A exists an unique morphism $I \to A$.

Definition 9 (Terminal object)**.** An object T is a **terminal object** if every object is the codomain of exactly one morphism from it. That is, for every object A exists an unique $A \to T$.

Definition 10 (Zero object)**.** A **zero object** is an object which is both initial and terminal at the same time.

Proposition 3 (Initial and final objects are essentially unique)**.** *Initial and final objects in a category are essentially unique; that is, any two initial objects are isomorphic and any two final objects are isomorphic.*

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

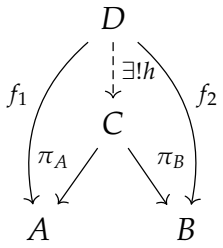
- $f \circ g = \mathrm{id}$,
- $g \circ f = \mathrm{id}$.

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

Definition 11 (Product object)**.** An object C is the **product** of two objects A, B on a category if there are two morphisms

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

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



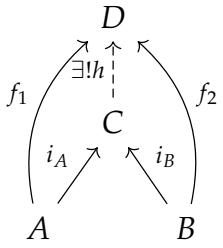
.

Note that the product of two objects does not have to exist on a category; but when it exists, it is essentially unique. In fact, we will be able later to construct a category in which the product object is the final object of the category and Proposition 3 can be applied. We will write *the* product object of A, B as $A \times B$.

Definition 12 (Coproduct object)**.** An object C is the **coproduct** of two objects A, B on a category if there are two morphisms

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

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



.

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

1.4 EXAMPLES OF CATEGORIES

Example 1 (Discrete categories). A category is **discrete** if it has no other morphisms than the identities. A discrete category is uniquely defined by its class of objects and every class of objects defines a category.

Example 2 (Monoids, groups). A single-object category is a **monoid**. [3] A monoid in which every morphism is an isomorphism is a **group**. A **groupoid** is a category (of any number of objects) where all the morphisms are isomorphisms.

Example 3 (Partially ordered sets). Every partial ordering defines a category in which the elements are the objects and an only morphism between two objects $\rho_{a,b} : a \to b$ exists

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

Example 4 (The category of sets). The category `Sets` is defined as the category with all the possible sets as objects and functions between them as morphisms. It is trivial to check associativity of composition and the existence of the identity function for any set.

Example 5 (The category of groups). The category `Grp` is defined as the category with groups as objects and group homomorphisms between them as morphisms.

Example 6 (The category of R-modules). The category R-Mod is defined as the category with R-modules as objects and module homomorphisms between them as morphisms. We know that the composition of module homomorphisms and the identity are also module homomorphisms.

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

Example 7 (The category of topological spaces). The category `Top` is defined as the category with topologicaal spaces as objects and continuous functions between them as morphisms.

3 : This definition is equivalent to the usual definition of monoid if we take the morphisms as elements of the monoid and composition of morphisms as the monoid operation.

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*.

Functors and natural transformations were defined for the first time by Eilenberg and MacLane in [EM42] while studying Čech cohomology. While initially they were devised mainly as a language for studying homology, they have proven its foundational value with the passage of time.

2.1 FUNCTORS

Definition 13 (Functor)**.** A **functor** will be interpreted as a morphism of categories. Given two categories $\mathcal{C}$ and $\mathcal{D}$, a functor between them $F : \mathcal{C} \to \mathcal{D}$ is given by

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

such that

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

Functors can be composed as we did with morphisms. In fact, a category of categories can be defined; having functors as morphisms.

Definition 14 (Composition of functors)**.** Given two functors $F \colon \mathcal{C} \to \mathcal{B}$ and $G \colon \mathcal{B} \to \mathcal{A}$, their composite functor $G \circ F : \mathcal{C} \to \mathcal{A}$ is given by the composition of object and arrow functions of the functors. This composition is trivially associative.

Definition 15 (Identity functor)**.** The identity functor on a category $I_{\mathcal{C}} \colon \mathcal{C} \to \mathcal{C}$ is given by identity object and arrow functions. It is trivially neutral with respect to composition.

Definition 16 (Full functor)**.** A functor F is **full** if the arrow map is surjective. That is, if every $g : FA \to FB$ is of the form Ff for some morphism $f : A \to B$.

Definition 17 (Faithful functor)**.** A functor F is **faithful** if the arrow map is injective. That is, if, for every two arrows $f_1, f_2 : A \to B$, $Ff_1 = Ff_2$ implies $f_1 = f_2$.

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

Definition 18 (Isomorphism of categories)**.** An **isomorphism of categories** is a functor T whose object and arrow functions are bijections. Equivalently, it is a functor T such that there exists an *inverse* functor S such that $T \circ S$ and $S \circ T$ are identity functors.

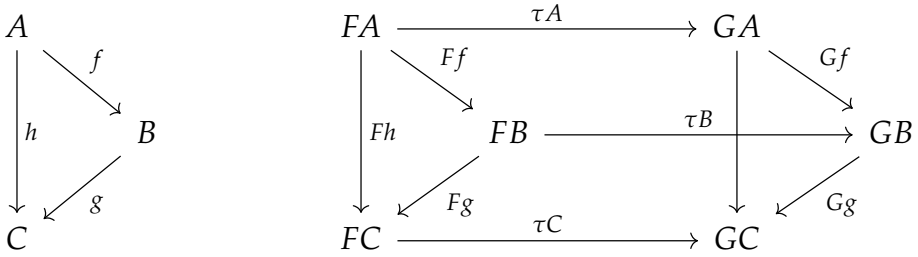
2.2 NATURAL TRANSFORMATIONS

Definition 19 (Natural transformation)**.** A **natural transformation** between two functors with the same domain and codomain, $\alpha : F \dot{\to} G$, is a family of morphisms parameterized by the objects of the domain category, $\alpha_C : FC \to GC$ such that the following diagram commutes

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

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

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

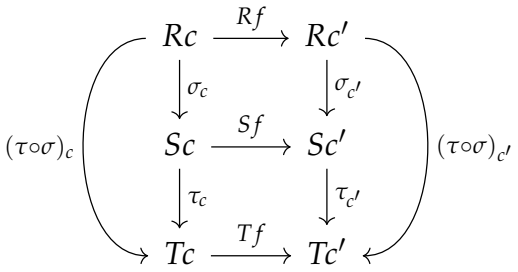


Definition 20 (Natural isomorphism)**.** A **natural isomorphism** is a natural transformation in which every component, every morphism of the parameterized family, is invertible.

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

2.3 COMPOSITION OF NATURAL TRANSFORMATIONS

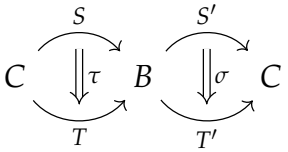
Definition 21 (Vertical composition of natural transformations)**.** The **vertical composition** of two natural transformations $\tau : S \to T$ and $\sigma : R \to S$, denoted by $\tau \cdot \sigma$ is the family of morphisms defined by the objectwise composition of the components of the two natural transformations, i.e.



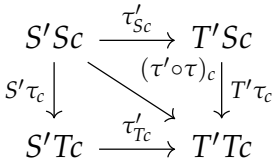
Proposition 4 (Vertical composition is a natural transformation)**.** *The vertical composition of two natural transformations is in fact a natural transformation.*

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

Definition 22 (Horizontal composition of natural transformations)**.** The **horizontal composition** of two natural transformations $\tau\colon S \to T$ and $\tau'\colon S' \to T'$, with domains and codomains as in the following diagram



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



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

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

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

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

3

CONSTRUCTIONS ON CATEGORIES

3.1 OPPOSITE CATEGORIES AND CONTRAVARIANT FUNCTORS

Definition 23 (Opposite category)**.** The **opposite category** $\mathcal{C}^{op}$ of a category $\mathcal{C}$ is a category with the same objects as $\mathcal{C}$ but with all its arrows reversed. That is, for each morphism $f : A \to B$, there exists a morphism $f^{op} : B \to A$ in $\mathcal{C}^{op}$. Composition is defined as

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

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

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

Definition 24 (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.

3.2 PRODUCT CATEGORIES

Definition 25 (Product category)**.** The **product category** of two categories $\mathcal{C}$ and $\mathcal{D}$, denoted by $\mathcal{C} \times \mathcal{D}$ is a category having

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

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

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

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

Definition 26 (Product of functors)**.** The **product functor** of two functors $F\colon \mathcal{C} \to \mathcal{C}'$ and $G\colon \mathcal{D} \to \mathcal{D}'$ is a functor $F \times G\colon \mathcal{C} \times \mathcal{D} \to \mathcal{C}' \times \mathcal{D}'$ which can be defined

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

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

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

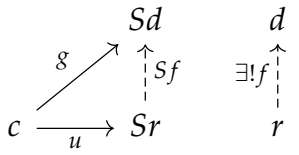
In this sense, the $\times$ operation is itself a functor acting on objects and morphisms of the `Cat` category of all categories.

4

UNIVERSALITY

4.1 UNIVERSAL ARROWS

Definition 27 (Universal arrow)**.** A **universal arrow** from c to S is an arrow $u\colon c \to Sr$ such that for every $c \to Sd$ exists a unique $r \to d$ making this diagram commute



.

Proposition 6 (Universality in terms of hom-sets)**.** *The arrow $u\colon c \to Sr$ is universal iff $f \mapsto Sf \circ u$ is a bijection $\hom(r,d) \cong \hom(c,Sd)$ natural in d. Any natural bijection of this kind is determined by a unique universal arrow.*

Proof. Bijection follows from the definition of universal arrow, and naturality follows from $S(gf) \circ u = Sg \circ Sf \circ u$.

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

4.2 REPRESENTABILITY

Definition 28 (Representation of a functor)**.** A **representation** of $K\colon D \to \texttt{Sets}$ is a natural isomorphism

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

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

Proposition 7 (Representations in terms of universal arrows)**.** *If $u\colon * \to Kr$ is a universal arrow for a functor $K\colon D \to \texttt{Sets}$, then $f \mapsto K(f)(u*)$ is a representation. Every representation is obtained in this way.*

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

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

for every natural isomorphism $D(r, -) \cong \mathtt{Sets}(*, K-)$. But every natural isomorphism of this kind is a [BROKEN LINK: *Universal arrows as natural bijections]. □

4.3 YONEDA LEMMA

Lemma 1 (Yoneda Lemma)**.** *For any $K\colon D \to \mathtt{Sets}$ and $r \in D$, there is a bijection*

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

sending the natural transformation $\alpha\colon \hom_D(r, -) \dot{\to} K$ to the image of the identity, $\alpha_r 1_r$.

Proof. □

Corollary 1 (Characterization of natural transformations between representable functors)**.** *Given $r, s \in D$, any natural transformation $\hom(r, -) \dot{\to} \hom(s, -)$ has the form h_* for a unique $h\colon s \to r$.*

Proof. Using Yoneda Lemma, we know that

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

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

Proposition 8 (Addendum to the Yoneda Lemma)**.** *The bijection on the Yoneda Lemma is a natural isomorphism between two $\mathtt{Sets}^D \times D \to \mathtt{Sets}$ functors.*

Proof. □

Definition 29. In the conditions of Yoneda Lemma, the **Yoneda functor**, $Y\colon D^{op} \to \mathtt{Sets}^D$, is defined with the arrow function

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

Proposition 9. *The Yoneda functor is full and faithful.*

Proof. □

5

ADJOINTS

5.1 ADJUNCTIONS

Definition 30 (Adjunction)**.** An **adjunction** from categories X to A is a pair of functors $F\colon X \to A$, $G\colon A \to X$ with a natural bijection

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

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

Definition 31 (Unit and counit of an adjunction)**.** An adjunction determines a **unit** and a **counit**;

1. the **unit** is natural transformation made with universal arrows $\eta\colon I \dot{\to} GF$, where the right adjoint of each $f\colon Fx \to a$ is

$$\varphi f = Gf \circ \eta_x\colon x \to Ga.$$

2. the **counit** is natural transformation made with universal arrows $\varepsilon\colon FG \dot{\to} I$, where the left adjoint of each $g\colon x \to Ga$ is

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

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

Proposition 10 (Characterization of adjunctions)**.** *Each adjunction is completely determined by any of*

1. *functors F, G and $\eta\colon 1 \dot{\to} GF$ where $\eta_x\colon x \to GFx$ is universal to G.*
2. *functor G and universals $\eta_x\colon x \to GF_0x$, creating a functor F.*
3. *functors F, G and $\varepsilon\colon FG \dot{\to} 1$ where $\varepsilon_a\colon FGa \to a$ is universal from F.*
4. *functor F and universals $\varepsilon_a\colon FG_0a \to a$, creating a functor G.*
5. *functors F, G, with units and counits satisfiying the triangle identities $\eta G \circ G\varepsilon = \mathrm{id}$ and $F\eta \circ \varepsilon F = \mathrm{id}$.*

Proof. □

6

MONADS AND ALGEBRAS

6.1 MONADS

Definition 32 (Monad)**.** A **monad** is a functor $T\colon X \to X$ with natural transformations

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

such that

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

Proposition 11 (Each adjunction gives rise to a monad)**.** *Given $F \dashv G$, GF is a monad.*

Proof. We take the unit of the adjunction as the monad unit. We define the product as $\mu = G\varepsilon F$. Associativity follows from these diagrams

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

where the first is commutative by the [BROKEN LINK: *Interchange law] and the second is obtained by applying functors G and F. Unit laws follow from the [BROKEN LINK: *Unit and counit] after applying F and G. □

Definition 33 (Comonad)**.** A **comonad** is a functor $L\colon X \to X$ with natural transformations

- $\varepsilon\colon L \to I$, called *counit*
- $\delta\colon L \to L^2$, called *comultiplication*

such that

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

6.2 ALGEBRAS FOR A MONAD

Definition 34 (T-algebra)**.** For a monad T, a **T-algebra** is an object x with an arrow $h\colon Tx \to x$ called *structure map* making these diagrams commute

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

Definition 35 (Morphism of T-algebras)**.** A **morphism of T-algebras** is an arrow $f\colon x \to x'$ making the following square commute

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

Proposition 12 (Category of T-algebras)**.** *The set of all T-algebras and their morphisms form a category X^T.*

Proof. Given $f\colon x \to x'$ and $g\colon x' \to x''$, T-algebra morphisms, their composition is also a T-algebra morphism, due to the fact that this diagram

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

commutes. □

Part II

LAMBDA CALCULUS

7

UNTYPED λ-CALCULUS

The **λ-calculus** is a collection of formal systems, all of them based on the lambda notation discovered by Alonzo Church in the 1930s while trying to develop a foundational notion of function on mathematics.

The **untyped** or **pure lambda calculus** is, syntactically, the simplest of those formal systems. This presentation of the untyped lambda calculus will follow [HS08] and [Sel13].

7.1 DEFINITION

Definition 36 (Lambda terms)**.** The **λ-terms** are defined inductively as

- every *variable*, taken from an infinite and numerable set $\mathcal{V}$ of variables, and usually written as lowercase single letters (x,y,z,...), is a λ-term.
- given two λ-terms M, N; its *application*, MN is a λ-term.
- given a λ-term M and a variable x, its *abstraction*, $\lambda x.M$ is a lambda term.

They can be also defined by the following BNF

$$\texttt{Exp} ::= x \mid (\texttt{Exp}\ \texttt{Exp}) \mid (\lambda x.\texttt{Exp})$$

where $x \in \mathcal{V}$ is any variable.

By convention, we omit outermost parentheses and assume left-associativity, i.e., MNP will mean $(MN)P$. Multiple λ-abstractions can be also contracted to a single multivariate abstraction; thus $\lambda x.\lambda y.M$ can become $\lambda x, y.M$.

7.2 FREE AND BOUND VARIABLES, SUBSTITUTION

Any ocurrence of a variable x inside the *scope* of a lambda is said to be bound; and any not bound variable is said to be free. We can define formally the set of free variables as follows.

Definition 37 (Free variables)**.** The **set of free variables** of a term M is defined inductively as

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

A free ocurrence of a variable can be substituted by a term. This should be done avoiding the unintended bounding of free variables which happens when a variable is substituted inside of the scope of a binder with the same name, as in the following example, where we substitute y by $(\lambda z.xz)$ on $(\lambda x.yx)$ and the second free variable x gets bounded by the first binder

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

To avoid this, the x should be renamed before the substitution.

Definition 38 (Substitution on lambda terms)**.** The **substitution** of a variable x by a term N on M is defined inductively as

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

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

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

7.3 α-EQUIVALENCE

Definition 39 ($\alpha - equivalence$)**.** **α-equivalence** is the smallest relation $=_\alpha$ on λ-terms which is an equivalence relation, i.e.,

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

and it is compatible with the structure of lambda terms,

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

α-equivalence formally captures the fact that the name of a bound variable can be changed without changing the properties of the term. This idea appears recurrently on mathematics; e.g., the renaming of the variable of integration is an example of α-equivalence.

$$\int_0^1 x^2 \, dx = \int_0^1 y^2 \, dy$$

7.4 β-REDUCTION

The core idea of evaluation in λ-calculus is captured by the notion of β-reduction.

Definition 40 (Beta-reduction)**.** The **single-step β-reduction** is the smallest relation on λ-terms capturing the notion of evaluation

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

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

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

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

7.5 η-REDUCTION

The idea of function extensionality in λ-calculus is captured by the notion of η-reduction. Function extensionality implies the equality of any two terms that define the same function over any argument.

Definition 41 (Eta reduction)**.** The η-reduction is the smallest relation on λ-terms satisfiying the same congruence rules as β-reduction and the following axiom

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

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

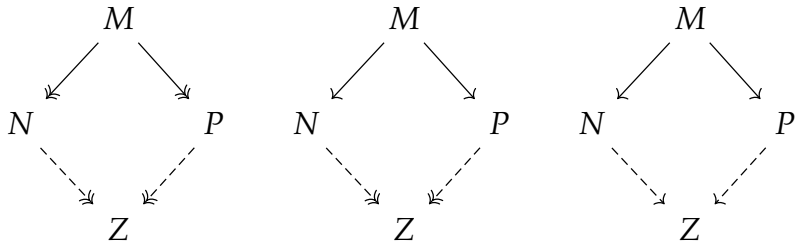
7.6 CONFLUENCE

Definition 42. A relation $\to$ is **confluent** if, given its reflexive transitive closure $\twoheadrightarrow$, $M \twoheadrightarrow N$ and $M \twoheadrightarrow P$ imply the existence of some Z such that $N \twoheadrightarrow Z$ and $P \twoheadrightarrow Z$.

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

- the **confluence** or Church-Rosser property we have just defined.
- the **quasidiamond property**, which assumes $M \to N$ and $M \to P$.
- the **diamond property**, which is defined substituting $\twoheadrightarrow$ by $\to$ on the definition on confluence.

Diagrammatically, the three properties can be represented as



and the implication relation between them is that the diamond relation implies confluence; while the quasidiamond does not. Both claims are easy to prove, and they show us that, in order to prove confluence for a given relation, we need to prove the diamond property instead of try to prove it from the quasidiamond property, as a naive attempt of proof would try.

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

7.7 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. It is based on the idea of parallel one-step reduction.

Definition 43 (Parallel one-step reduction)**.** We define the **parallel one-step reduction** relation, $\rhd$ as the smallest relation satisfying that, assuming $P \rhd P'$ and $N \rhd N'$, the following properties of

- reflexivity, $x \rhd x$;

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

hold.

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

Lemma 2. *The reflexive transitive closure of $\rhd$ is $\twoheadrightarrow_{\beta\eta}$. In particular, given any M, M',*

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

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

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

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

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

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

Lemma 3 (Substitution Lemma)**.** *Assuming $M \rhd M'$ and $U \rhd U'$, $M[U/y] \rhd M'[U'/y]$.*

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

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

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

Definition 44 (Maximal parallel one-step reduct)**.** The **maximal parallel one-step reduct** M^* of a λ-term M is defined inductively as

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

Lemma 4 (Diamond property of parallel reduction)**.** *Given any M' such that $M \rhd M'$, $M' \rhd M^*$. Parallel one-step reduction has the diamond property.*

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

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

- Parallel substitution, with $N \rhd N'$ and $Q \rhd Q'$; we know that $M^* = Q^*[N^*/x]$ and we can apply the substitution lemma (lemma 3) to get $M' \rhd M^*$.
- Extensionality. We know that $P \rhd P'$ and $x \notin \mathrm{FV}(P)$, so by induction hypothesis we know that $P' \rhd P^* = M^*$.

□

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

Proof. Parallel reduction, $\rhd$, satisfies the diamond property (lemma 4), which implies the Church-Rosser property. Its reflexive transitive closure is $\twoheadrightarrow_{\beta\eta}$ (lemma 2), whose diamond property implies confluence for $\rightarrow_{\beta\eta}$. □

7.8 NORMALIZATION

Definition 45 (Normal forms)**.** A λ-term is said to be in **β-normal form** if β-reduction cannot be applied to it or any of its subformulas. We define **η-reduction** and **$\beta\eta$-reduction** analogously.

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

Definition 46. A term is **weakly normalizing** if there exists a sequence of reductions from it to a normal form. It is **strongly** normalizing if every sequence of reductions is finite.

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

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

is neither weakly nor strongly normalizing; and the term

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

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

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

7.9 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 this strategy. Our basic result will be the **standarization**

theorem, which shows that, if a β-reduction to a normal form exists, a sequence of β-reductions from left to right on the λ-expression will be able to find it. From this result, we will be able to prove that the reduction strategy that always reduces the leftmost β-abstraction will always find a normal form if it exists.

This section follows [Kaso0], [Bar94] and [Bar84].

Definition 47 (Leftmost one-step reduction)**.** We define the relation $M \to_n N$ when N can be obtained by β-reducing the n-th leftmost β-reducible application of the expression. We call $\to_1$ the **leftmost one-step reduction** and we write it as $\to_l$; $\twoheadrightarrow_l$ is its reflexive transitive closure.

Definition 48 (Standard sequence)**.** A sequence of β-reductions $M_0 \to_{n_1} M_1 \to_{n_2} M_2 \to_{n_3} \cdots \to_{n_k} M_k$ is **standard** if $\{n_i\}$ is a non-decreasing sequence.

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

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

Proof. We start by defining the following two binary relations. The first one is the relation given by the head reduction of the application and it is defined by

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

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

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

We can check the following trivial properties by structural induction

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

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

$P \twoheadrightarrow_s \lambda x.M$ and $Q \twoheadrightarrow_s N$; and from $P \twoheadrightarrow_s \lambda x.M$, we know that there exists W such that $P \twoheadrightarrow_h \lambda x.W$ and $W \twoheadrightarrow_s M$. From all this information, we can conclude that

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

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

We finally prove that, if $K \twoheadrightarrow_s M \to_\beta N$, then $K \twoheadrightarrow_s N$. This proves the theorem, as every β-reduction $M \twoheadrightarrow_s M \to_\beta N$ 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 2 (Leftmost reduction theorem)**.** *We define the **leftmost reduction strategy** as the strategy that reduces the leftmost β-reducible application at each step. If M has a normal form, the leftmost reduction strategy will lead to it.*

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

7.10 SKI COMBINATORS

Definition 49 (Lambda transform)**.** The **Λ-transform** of a Ski-term is a λ-term defined recursively as

- $\Lambda(x) = x$, for any variable x;
- $\Lambda(I) = (\lambda x.x)$;
- $\Lambda(K) = (\lambda x.\lambda y.x)$;
- $\Lambda(S) = (\lambda x.\lambda y.\lambda z.xz(yz))$;
- $\Lambda(XY) = \Lambda(X)\Lambda(Y)$.

Definition 50 (Bracket abstraction)**.** The **bracket abstraction** of the Ski-term U on the variable x is written as $[x].U$ and defined recursively as

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

- $[x].UV = S([x].U)([x].V)$, otherwise.

where FV is the set of free variables; as defined on Definition 37.

Definition 51 (Ski abstraction)**.** The **SKI abstraction** of a λ-term M, written as $\mathfrak{H}(M)$ is defined recursively as

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

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

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

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

Proof. By induction on U,

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

□

8

SIMPLY TYPED λ-CALCULUS

We will give now a presentation of the **simply-typed λ-calculus** (STLC) based on [HS08]. Our presentation will rely only on the *arrow type* $\to$; while other presentations of simply typed λ-calculus extend this definition with type constructors such as pairs or union types, as it is done in [Sel13].

It seems clearer to present a first minimal version of the λ-calculus. Such extensions will be explained later, profiting from the logical interpretation of propositions as types.

8.1 SIMPLE TYPES

We start assuming that a set of **basic types** exists. Those basic types would correspond, in a programming language interpretation, with things like the type of strings or the type of integers.

Definition 52 (Simple types)**.** The set of **simple types** is given by the following Backus-Naur form

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

where ι would be any *basic type*.

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

8.2 TYPING RULES FOR THE SIMPLY TYPED λ-CALCULUS

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

Definition 53 (Raw typed lambda terms)**.** The set of **typed lambda terms** is given by the BNF

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

The set of raw typed λ-terms contains some meaningless terms under our type interpretation, such as $\pi_1(\lambda x^A.M)$. **Typing rules** will give them the desired semantics; only a subset of these raw lambda terms will be typeable.

Definition 54 (Typing context)**.** A **typing context** is a sequence of typing assumptions $x_1 : A_1, \ldots, x_n : A_n$, where no variable appears more than once.

Every typing rule assumes a typing context, usually denoted by Γ or by a concatenation of typing contexts written as Γ, Γ'; and a consequence from that context, separated by the $\vdash$ symbol.

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

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

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

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

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

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

Definition 55. A term is **typeable** if we can assign types to all its variables in such a way that a typing judgment for the type is derivable.

From now on, we only consider typeable terms on the STLC to be used as real terms. As a consequence, the set of λ-terms of the STLC is only a subset of the terms of the untyped λ-calculus.

8.3 CURRY-STYLE TYPES

Two different approaches to typing λ-terms are commonly used.

- **Church-style** typing, also known as *explicit typing* originated from the work of Alonzo Church in [Chu40], where he described a STLC with two basic types. The term's type is defined as an intrinsic property of the term; and the same term has to be interpreted always with the same type.
- **Curry-style** typing, also known as *implicit typing*; which creates a formalism where every single term can be given an infinite number of types. This technique is called **polymorphism** in general; and here it is only used to tell the kinds of combinations that are allowed with any given term.

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

Definition 56 (Type variables)**.** Given a infinite numerable set of *type variables*, we define **parametric types** or *type-schemes* inductively as

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

that is, all basic types and type variables are atomic parametric types; and we also consider the arrow type between two parametric types.

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

Definition 57 (Type substitution)**.** A **substitution** ψ is any function from type variables to type templates. It can be applied to a type template as $\overline{\psi}$ by recursion and knowing that

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

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

We consider a type to be *more general* than other if the latter can be obtained by applying a substitution to the former. In this case, the latter is called an *instance* of the former. For instance, $A \to B$ is more general than its instance $(C \to D) \to B$, where A is a type variable affected by the substitution. An interesting property of STLC is that every type has a most general type, called its *principal type*.

Definition 58 (Principal type)**.** A closed λ-term M has a **principal type** π if $M : \pi$ and given any $M : \tau$, we can obtain τ as an instance of π, that is, $\overline{\sigma}\pi = \tau$.

8.4 UNIFICATION AND TYPE INFERENCE

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

Definition 59 (Most general unifier)**.** A substitution ψ is called an **unifier** of two sequences of type templates $\{A_i\}_{i=1,\dots,n}$ and $\{B_i\}_{i=1,\dots,n}$ if $\overline{\psi}A_i = \overline{\psi}B_i$ for any i. We say

that it is the **most general unifier** if given any other unifier ϕ exists a substitution φ such that $\phi = \overline{\varphi} \circ \psi$.

Lemma 5. *If an unifier of $\{A_i\}_{i=1,\dots,n}$ and $\{B_i\}_{i=1,\dots,n}$ exists, the most general unifier can be found using the following recursive definition of* $\texttt{unify}(A_1, \dots, A_n; B_1, \dots, B_n)$.

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

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

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

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

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

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

Using unification, we can define type inference.

Theorem 4 (Type inference)**.** *The algorithm* $\texttt{typeinfer}(M, B)$*, defined as follows, finds the most general substitution* σ *such that* $x_1 : \sigma A_1, \ldots, x_n : \sigma A_n \vdash M : \overline{\sigma} B$ *is a valid typing judgment if it exists; and fails otherwise.*

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

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

Proof. The algorithm terminates by induction on the size of M. It is easy to check by structural induction that the inferred type judgments are in fact valid. If the algorithm fails, by Lemma 5, 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 5. 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 3 (Principal type property)**.** *Every typeable pure* λ*-term has a principal type.*

Proof. Given a typeable term M, we can compute $\texttt{typeinfer}(x_1 : A_1, \ldots, x_n : A_n \vdash M : B)$, where $x_1, \ldots, x_n$ are the free variables on M and $A_1, \ldots, A_n, B$ are fresh type variables. By virtue of Theorem 4, the result is the most general type of M if we assume the variables to have the given types. □

9

THE CURRY-HOWARD CORRESPONDENCE

9.1 EXTENDING THE SIMPLY TYPED λ-CALCULUS

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

Definition 60 (Simple types II)**.** The new set of **simple types** is given by the following BNF

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

where ι would be any *basic type*.

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

Definition 61 (Raw typed lambda terms II)**.** The new set of raw **typed lambda terms** is given by the BNF

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9.2 NATURAL DEDUCTION

The natural deduction is a logical system due to Gentzen. We introduce it here following [Sel13] and [Wad15]. It relationship with the STLC will be made explicit on the next section.

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

1. Every axiom on the context can be used.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9.3 PROPOSITIONS AS TYPES

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

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

3. simplification of proofs is the evaluation of programs.

In the case of STLC and natural deduction, the correspondence starts when we describe the following isomorphism between types and propositions.

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

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

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

Example 8 (Curry-Howard example). As an example, we will write a proof/term of the proposition/type `A → B + A` and we are going to simplify/compute it using proof simplification rules/β-rules.

We start with the following derivation tree

$$\dfrac{\dfrac{[A+B] \quad \dfrac{\dfrac{[A]}{B+A}\,(inr) \quad \dfrac{[B]}{B+A}\,(inl)}{B+A}\,(case)}{A+B \to B+A}\,(abs) \qquad \dfrac{[A]}{A+B}\,(inl)}{\dfrac{B+A}{A \to B+A}\,(abs)}\,(app)$$

which is encoded by the term `λa.(λc.case c of (λx.inr) (λy.inl y)) (inl a)`. We apply the simplification rule/β-rule of the implication/function application to get

$$\dfrac{\dfrac{[A]}{A+B}\,(inl) \quad \dfrac{[A]}{B+A}\,(inr) \quad \dfrac{[B]}{B+A}\,(inl)}{B+A}\,(case)}{A \to B+A}\,(abs)$$

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

$$\frac{\dfrac{[A]}{B+A}\,(inr)}{A \to B+A}\,(abs)$$

which is encoded by `λa.(inr a)`.

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

```
\a.((\c.caseof c (\x.inr x) (\y.inl y))(inl a))
---> output: λa.(INR a) ⇒ inr :: A → B + A
```

10

OTHER TYPE SYSTEMS

10.1 λ-CUBE

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

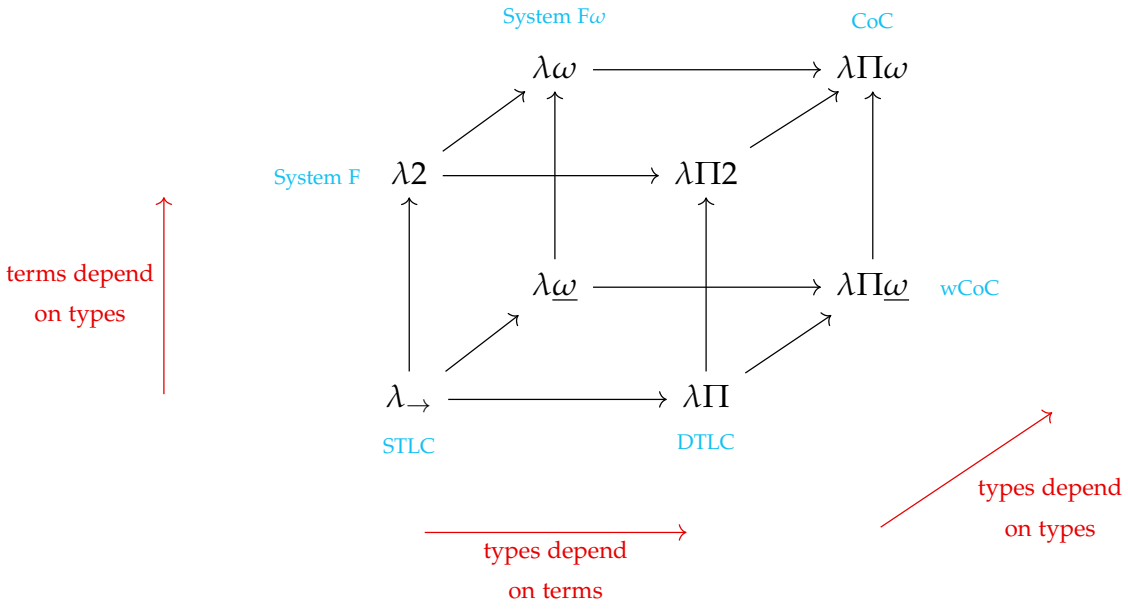
1. **parametric polymorphism**, terms that depend on types. This is achieved via universal quantification over types. It allows type variables and binders for them as in the following parametric identity function

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

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

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

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



The following type systems

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

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

All systems on the λ-cube are strongly normalizing.

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

Part III

MIKROKOSMOS

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

11

PROGRAMMING ENVIRONMENT

11.1 THE HASKELL PROGRAMMING LANGUAGE

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

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

The complete history of Haskell and its design decisions is detailed on [HHJW07]. Haskell is

- **strongly and statically typed**, meaning that it only compiles well-typed programs and it does not allow implicit type casting. The compiler will generate an error if a term is non-typeable.
- **lazy**, with *non-strict semantics*, meaning that
  This *call-by-need* approach is usually less efficient than the *call-by-value* one.
  In [Hughes Why functional programming matters], Hughes argued for the benefits of a lazy functional language.
- **purely functional**. As the evaluation order is demand-driven and not explicitly known, it is not possible in practice to perform ordered input/output actions or

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

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

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

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

```haskell
-- A tree is either empty or a node with two subtrees.
data Tree a = Empty | Node a (Tree a) (Tree a)

-- The preorder function takes a tree and returns a list
preorder :: Tree a -> [a]
preorder Empty            = []
preorder (Node x lft rgt) = preorder lft ++ [x] ++ preorder rgt
```

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

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

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

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

where the second one, because of its simplicity, is usually preferred. A characteristic piece of Haskell are **type classes**, which allow to define common interfaces for different

types. In the following example, we define a `Monad` as a type with a suitably typed `return` and `bind` operators.

```
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]
  xs >>= f = concat (map f xs)
```

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

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

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

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

11.2 VERSION CONTROL AND CONTINUOUS INTEGRATION

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

`https://github.com/M42/mikrokosmos`

Development takes place on the `development` git branch and permanent changes are released into the `master` branch.

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

12

IMPLEMENTATION OF λ-EXPRESSIONS

12.1 DE BRUIJN INDEXES

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

Consider the following example, the λ-term

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

can be written with de Bruijn indices as

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

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

Definition 62 (De Bruijn indexed terms)**.** We define recursively the set of λ-terms using de Bruijn notation following this BNF

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

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

```haskell
-- | A lambda expression using DeBruijn indexes.
data Exp = Var Integer -- ^ integer indexing the variable.
```

```
        | Lambda Exp  -- ^ lambda abstraction
        | App Exp Exp -- ^ function application
        deriving (Eq, Ord)
```

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

12.2 SUBSTITUTION

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

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

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

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

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

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

12.3 DE BRUIJN-TERMS AND λ-TERMS

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

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

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

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

-- Translation of applications is trivial.
tobruijn d context (LambdaApplication f g) =
     App (tobruijn d context f) (tobruijn d context g)

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

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

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

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

12.4 EVALUATION

As we proved on Corollary 2, the leftmost reduction strategy will find the leftmost reduction strategy if it exists. Consequently, we will implement it using a function that simply applies the leftmost possible reductions at each step. This will allow us to show how the interpreter performs step-by-step evaluations to the final user, as discussed in the verbose mode section.

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

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

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

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

12.5 PRINCIPAL TYPE INFERENCE

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

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

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

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

```haskell
type Substitution = Type -> Type

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

Unification will be implemented making extensive use of the `Maybe` monad. If the unification fails, it will return an error value, and the error will be propagated to all the computation. The algorithm is exactly the same that was defined in Lemma 5.

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

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

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

```haskell
-- | Type inference algorithm. Infers a type from a given context and expression
-- with a set of constraints represented by a unifier type. The result type must
-- be unifiable with this given type.
typeinfer :: [Variable] -- ^ List of fresh variables
          -> Context    -- ^ Type context
          -> Exp        -- ^ Lambda expression whose type has to be inferred
          -> Type       -- ^ Constraint
          -> Maybe Substitution

typeinfer (x:vars) ctx (App p q) b = do -- Writing inside the Maybe monad.
  sigma <- typeinfer (evens vars) ctx                 p (Arrow (Tvar x) b)
  tau   <- typeinfer (odds  vars) (applyctx sigma ctx) q (sigma (Tvar x))
```

```
  return (tau . sigma)
  where
    -- The list of fresh variables has to be split into two
    odds [] = []
    odds [_] = []
    odds (_:e:xs) = e : odds xs
    evens [] = []
    evens [e] = [e]
    evens (e:_:xs) = e : evens xs
```

The final form of the type inference algorithm will use a normalization algorithm shortening the type names and will apply the type inference to the empty type context.

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

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

13

USER INTERACTION

13.1 MONADIC PARSER COMBINATORS

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

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

```haskell
newtype Parser a = Parser (String -> [(a,String)])

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

instance Monad Parser where
  return x = Parser (\s -> [(x,s)])
  p >>= q  = Parser (\s ->
              concat [parse (q x) s' | (x,s') <- parse p s ])
```

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

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

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

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

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

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

13.2 PARSEC

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

13.3 VERBOSE MODE

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

The difference can be seen on the following example.

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

mikro> :verbose on
verbose: on
```

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

λa.λb.(a (a (a b))) ⇒ 3
```

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

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

13.4 SKI MODE

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

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

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

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

mikro> and
λa.λb.((a b) a) ⇒ SSK ⇒ and
```

The code implementing these algorithms follows directly from the theoretical version.

```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)
```

14

USAGE

14.1 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; and
4. the **Mikrokosmos libraries**, written in the Mikrokosmos language.

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

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

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

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

1 : Hackage can be accesed in: http://hackage.haskell.org/

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

```
sudo pip install imikrokosmos
```

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

```
jupyter kernelspec list
```

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

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

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

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

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

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

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

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

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

14.2 MIKROKOSMOS INTERPRETER

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

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

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

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

The Figure 1 is an example session on the mikrokosmos interpreter.

14.3 JUPYTER KERNEL

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

We have developed a Mikrokosmos kernel for the Jupyter Notebook, allowing the user to write and execute arbitrary Mikrokosmos code on this web application. An example session can be seen on Figure 2.

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 = ""
```

Figure 1: Mikrokosmos interpreter session.

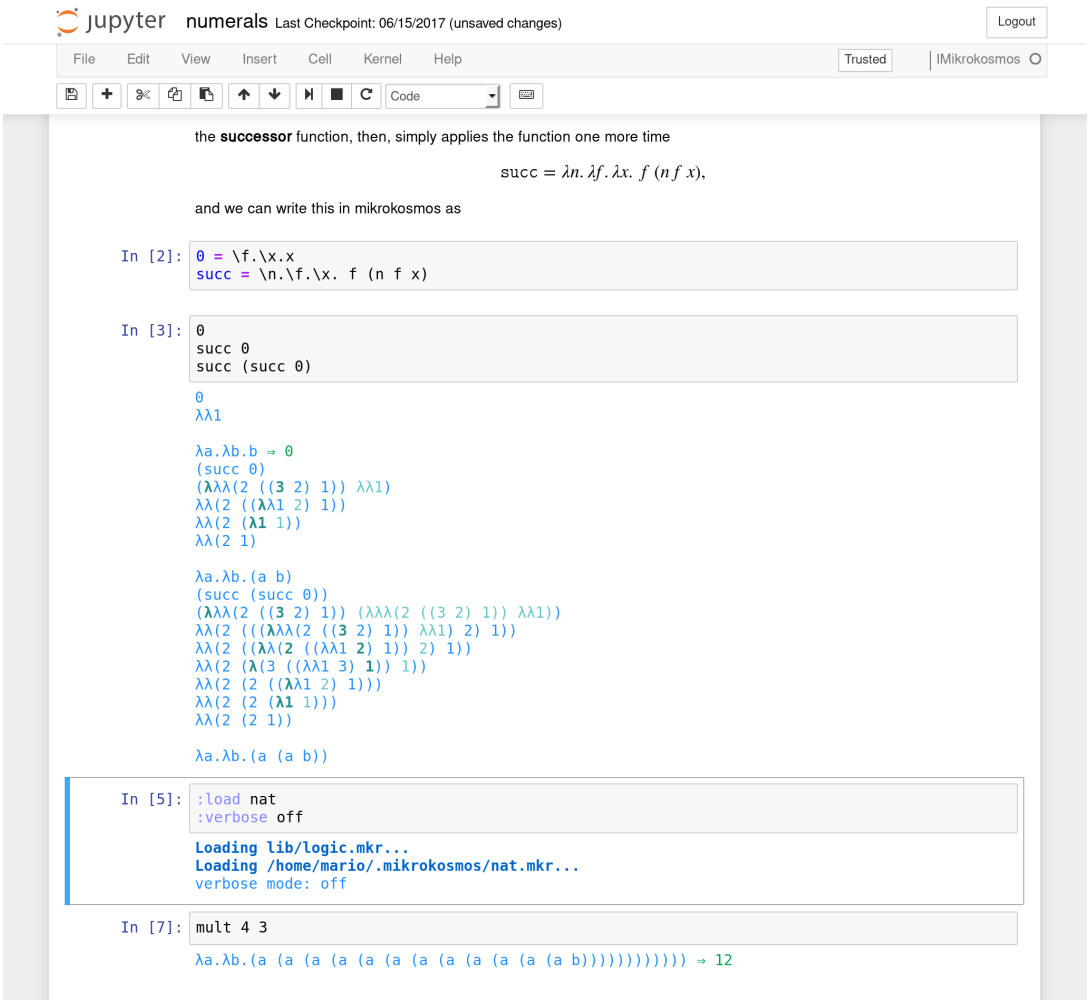


Figure 2: Jupyter notebook Mikrokosmos session.

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

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

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

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

14.4 CODEMIRROR LEXER

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

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

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

3 : The PyPA packaging user guide can be found in its official page: https://packaging.python.org/
4 : Documentation for CodeMirror can be found in its official page: https://codemirror.net/

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

15

PROGRAMMING IN THE UNTYPED λ-CALCULUS

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

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

All the code on this section is valid Mikrokosmos code.

15.1 BASIC SYNTAX

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

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

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

Evaluation of terms will be denoted with the => symbol, as in

```
compose id id
---> λa.a ⇒ id
```

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

```
alwaysid = const id
```

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

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

or the definition of exponentials.

15.2 A TECHNIQUE ON INDUCTIVE DATA ENCODING

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

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

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

Or, in general

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

We do not have any possibility of encoding constructors on λ-calculus. Even if we had, they would have, in theory, no computational content; the application of constructors would not be reduced under any λ-term, and we would need at least the ability to pattern match on the constructors to define functions on them. The λ-calculus would need to be extended with additional syntax for every new type.

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

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

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

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

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

15.3 BOOLEANS

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

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

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

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

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

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

We can test this interpretation on the interpreter to get

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

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

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

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

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

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

xor true true
--- => false
```

15.4 NATURAL NUMBERS

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

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

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

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

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

We can then name some natural numbers as

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

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

```
5 not true
--- => false
```

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

double = \n.\s.\z.n (compose s s) z
double 3
--- => 6
```

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

```
plus = \m.\n.\s.\z.m s (n s z)
mult = \m.\n.\s.\z.m (n s) z

plus 2 1
--- => 3
mult 2 4
--- => 8
```

15.5 THE PREDECESSOR FUNCTION AND PREDICATES ON NUMBERS

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

```
iszero = \n.(n (const false) true)
iszero 0
--- => true
iszero 2
--- => false
```

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

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

15.6 LISTS

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

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

Our definition takes those two constructors into account

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

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

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

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

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

sum (cons 1 (cons 2 (cons 3 nil)))
--- => 6
all (cons true (cons true (cons true nil)))
--- => true
```

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

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

They can be defined as follows.

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

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

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

16

PROGRAMMING IN THE SIMPLY TYPED λ-CALCULUS

This section explains how to use the simply typed λ-calculus to encode compound data structures and proofs on intuitionistic logic.

This presentation of simply typed structures follows the Mikrokosmos tutorial and the previous sections on simply typed λ-calculus.

All the code on this section is valid Mikrokosmos code.

16.1 FUNCTION TYPES AND TYPEABLE TERMS

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

Example 9 (Typed terms on Mikrokosmos). The following are examples of already defined terms on lambda calculus and their corresponding types. It is important to notice how our previously defined booleans have two different types; while our natural numbers will have all the same type except from zero, whose type is a generalization on the type of the natural numbers.

```
id
---> λa.a ⇒ id, I, ifelse :: A → A

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

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

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

S
K
---> λa.λb.λc.((a c) (b c)) ⇒ S :: (A → B → C) → (A → B) → A → C
---> λa.λb.a ⇒ K, true :: A → B → A
```

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

Example 10 (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.

16.2 PRODUCT, UNION, UNIT AND VOID TYPES

Until this point, we have only used the function type. We are working on the implicational fragment of the STLC we described on the first typing rules. We are now going to extend the type system in the same sense we extended the STLC. The following types are added to the type system

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

And the following typed constructors are added to the language,

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

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

Example 11 (Extended STLC on Mikrokosmos). The following are examples of typed terms and functions on Mikrokosmos using the extended typed constructors. The following terms are presented

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

```
:load types

swap = \m.(snd m,fst m)
swap
---> λa.((SND a),(FST a)) ⇒ swap :: (A × B) → B × A

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

\x.((abort x),(absurd x))
---> λa.((ABORT a),(ABSURD a)) :: ⊥ → A × ⊥
```

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

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

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

16.3 A PROOF ON INTUITIONISTIC LOGIC

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

Theorem 5. *In intuitionistic logic, the double negation of the LEM holds for every proposition, that is,*

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

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

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

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

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

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

Part IV

CATEGORICAL LOGIC

This section is based on [MM94].

Topos theory arises independently with Grothendieck and sheaf theory, Lawvere and the axiomatization of set theory and Paul Cohen with the forcing techniques with allowed to construct new models of ZFC.

17

LAWVERE ALGEBRAIC THEORIES

17.1 ALGEBRAIC THEORIES

Definition 63 (Algebraic theory)**.** An **algebraic theory** is given by

- a *signature*, a family of sets $\{\Sigma_k\}_{k\in\mathbb{N}}$ whose elements are called **k-ary operations**. The **terms** of a signature are defined inductively as variables or k-ary operations applied to k-tuples of terms.
- a set of **axioms**, which are equations between terms.

Definition 64 (Interpretation of an algebraic theory)**.** An **interpretation** I of a theory $\mathbb{A}$ on a category $\mathcal{C}$ is given by

- an object on the category, $I\mathbb{A} \in \mathcal{C}$;
- a morphism $If\colon (I\mathbb{A})^k \to I\mathbb{A}$ for every k-ary operation.

Given an interpretation of the theory we can given an interpretation to every term of the signature on a given variable context. A term t can be given in the variable context $x_1, \ldots, x_n$ if all variables that appear on t are among the variables of the context; we write that as

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

Given I, the **interpretation of a term** $x_1, \ldots, x_n \mid t$ is a morphism $It\colon (I\mathbb{A})^n \to I\mathbb{A}$ defined inductively as

- $I(x_i) = \pi_i : (I\mathbb{A})^n \to I\mathbb{A}$, the i-th projection.
- $I(f\langle t_1, \ldots, t_k\rangle) = If \circ \langle It_1, \ldots, It_k\rangle\colon (I\mathbb{A})^n \to \mathbb{A}$, where $\langle It_1, \ldots, It_k\rangle\colon (I\mathbb{A})^n \to (I\mathbb{A})^n$ is the componentwise interpretation of subterms.

Note that the interpretation of variables depends on the context. An interpretation **satisfies** an equation $u = v$ in a context if $Iu = Iv$.

Definition 65 (Model of an algebraic theory)**.** A **model of an algebraic theory** is an interpretation that satisfies all the axioms of the theory.

17.2 ALGEBRAIC THEORIES AS CATEGORIES

Definition 66 (Lawvere algebraic theory)**.** An **algebraic theory** is a category with finite products and objects forming a sequence $A^0, A^1, A^2, \ldots$ such that $A^m \times A^n = A^{m+n}$ for any m, n.

From this definition, it follows that A^0 must be the terminal object.

Every model M in the sense of Definition [TODO] can be seen as a functor from the category of $\mathbb{A}$ to a given category $\mathcal{C}$; defined on objects as

$$M[x_1, \ldots, x_n] = (M\mathbb{A})^k$$

and inductively defined on morphisms as

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

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

Definition 67 (Model)**.** A **model** of an algebraic theory $\mathbb{A}$ in a category $\mathcal{C}$ is a functor $M \colon \mathbb{A} \to \mathcal{C}$ preserving all finite products.

Definition 68 (Category of models of a theory)**.** The **category of models** $\mathtt{Mod}_{\mathcal{C}}(\mathbb{A})$ is is the full subcategory of functor category $\mathcal{C}^{\mathbb{A}}$ given by the functors preserving all finite products.

17.3 COMPLETENESS FOR ALGEBRAIC THEORIES

Theorem 6 (Completeness for algebraic theories)**.** *Given $\mathbb{A}$ an algebraic theory, there exists a category $\mathcal{A}$ with a model $U \in \mathtt{Mod}_{\mathcal{A}}(\mathbb{A})$ such that, for every terms u, v,*

$$U \text{ satisfies } u = v \iff \mathbb{A} \text{ proves } u = v$$

*this is called the **universal model** for $\mathbb{A}$. That is, categorical semantics of algebraic theories are complete.*

Proof. □

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

Proposition 13 (Yoneda embedding as a universal model)**.** *The Yoneda embedding $y \colon \mathbb{A} \to \widehat{\mathbb{A}}$ is a universal model for $\mathbb{A}$.*

Proof. □

18

HEYTING ALGEBRAS

In this section, we develop the notion of a **Heyting algebra** and show its differences with a Boolean algebra.

There is a correlation between classical propositional calculus and the Boolean algebra of the subsets of a given set. If we interpret a proposition p as a subset of a given universal set $P \subset U$ and fix an element $u \in U$, propositions can be translated to $u \in P$, logical connectives can be translated as

logic			subsets
$P \wedge Q$	and	intersection	$P \cap Q$
$P \vee Q$	or	union	$P \cup Q$
$\neg P$	not	complement	$\overline{P}$
$P \to Q$	implication	complement union	$\overline{P} \cup Q$

using crucially that $\neg P \wedge Q \equiv P \to Q$.

In the same way that Boolean algebras correspond to classical propositional logic, Heyting algebras correspond to intuitionistic propositional calculus. Its model on a set-like theory is not the subsets of a given set, but instead, only the *open* sets of a given topological space

logic			open sets
$P \wedge Q$	and	intersection	$P \cap Q$
$P \vee Q$	or	union	$P \cup Q$
$\neg P$	not	interior of the complement	$\text{int}\left(\overline{P}\right)$
$P \to Q$	implication	interior of complement and consequent	$\text{int}(\overline{P} \cup Q)$

where int is the topological interior of a set.

18.1 BOOLEAN ALGEBRAS

Definition 69 (Lattice)**.** A **lattice** is a partially ordered set with all binary products and coproducts. It is a **bounded lattice** if it has all binary products and coproducts.

We will usually work with bounded lattices. A bounded lattice can also be defined as a set with $0, 1$ and two binary operations $\wedge, \vee$ satisfying

- $1 \wedge x = x$, and $0 \vee x = x$;
- $x \wedge x = x$, and $x \vee x = x$;
- $x \wedge (y \vee x) = x = (x \wedge y) \vee x$.

This perspective allows us also to define a lattice object in any category as an object L with morphisms

$$\wedge \colon L \times L \to L, \quad \vee \colon L \times L \to L, \quad 0, 1 \colon \mathrm{I} \to L,$$

where I is the terminal object of the category; and commutative diagrams encoding the previous equations.

Definition 70 (Distributive lattice)**.** A **distributive lattice** is a lattice where

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z),$$

holds for all x, y, z.

Definition 71 (Complement)**.** A **complement** of a in a bounded lattice is an element $\overline{a}$ such that

$$a \wedge \overline{a} = 0, \qquad a \vee \overline{a} = 1.$$

Proposition 14 (The complement in distributive lattices is unique)**.** *If a complement of an element exists in a distributive lattice, it is unique.*

Proof. Given a with two complements x, y, we have that

$$x = x \wedge (a \vee y) = (x \wedge a) \vee (x \wedge y) = (y \wedge a) \vee (x \wedge y) = y \vee (x \wedge a) = y.$$

□

Definition 72 (Boolean algebra)**.** A **Boolean algebra** is a distributive bounded lattice in which every element has a complement.

Boolean algebras satisfy certain known properties such as the DeMorgan laws and the double negation elimination rule.

18.2 HEYTING ALGEBRAS

Definition 73 (Heyting algebra)**.** A **Heyting algebra**, also called **Brouwerian lattice**, is a bounded lattice which is cartesian closed as a category; i.e., for every pair of elements x, y, the exponential y^x exists.

The exponential in Heyting algebras is usually written as $x \Rightarrow y$ and is characterized by its adjunction with the product

$$z \leq (x \Rightarrow y) \text{ if and only if } z \wedge x \leq y.$$

Proposition 15 (Boolean algebras are Heyting algebras)**.** *Every Boolean algebra is a Heyting algebra with exponentials given by*

$$(x \Rightarrow y) = \overline{x} \wedge y.$$

Proof. We will prove that

$$z \leq (\overline{x} \vee y) \text{ if and only if } z \wedge x \leq y.$$

If $z \leq (\overline{x} \vee y)$,

$$z \wedge x \leq (\overline{x} \vee y) \wedge x \leq (\overline{x} \wedge x) \vee (y \wedge x) \leq y \wedge x \leq y;$$

and if $z \wedge x \leq y$,

$$z = z \wedge 1 = z \wedge (\overline{x} \vee x) = (z \wedge \overline{x}) \vee (z \wedge x) \leq (z \wedge \overline{x}) \vee y \leq z \vee y.$$

□

Definition 74 (Negation)**.** The **negation** of x in a Heyting algebra is defined as

$$\neg x = (x \Rightarrow 0).$$

Proposition 16. *In any Heyting algebra,*

1. $x \leq \neg\neg x$,
2. $x \leq y$ *implies* $\neg y \leq \neg x$,
3. $\neg x = \neg\neg\neg x$,
4. $\neg\neg(x \wedge y) = \neg\neg x \wedge \neg\neg y$,
5. $(x \Rightarrow x) = 1$,
6. $x \wedge (x \Rightarrow y) = x \wedge y$,
7. $y \wedge (x \Rightarrow y) = y$,
8. $x \Rightarrow (y \wedge z) = (x \Rightarrow y) \wedge (x \Rightarrow z)$.

Any bounded lattice L with an operation satisfying the last four properties is a Heyting algebra with this operation as implication.

Proof. We can prove the inequalities using the definition of implication.

1. By definition, $x \wedge (x \Rightarrow \bot) \leq \bot$.
2. Again, by definition, $\neg y \wedge x \leq \neg y \wedge y \leq \bot$.
3. Is a consequence of the first two inequalities.
4. We know that $x \wedge y \leq x, y$, and therefore $\neg\neg(x \wedge y) \leq \neg\neg x \wedge \neg\neg y$. We can prove $\neg\neg x \wedge \neg\neg y \leq \neg\neg(x \wedge y)$ using the definition of negation to get $\neg\neg x \wedge \neg\neg y \wedge \neg(x \wedge y) \leq \bot$, and then by reversing the definition of implication $\neg\neg y \wedge \neg(x \wedge y) \leq \neg\neg\neg x = x$. Applying the same reasoning to y, we finally get $x \wedge y \wedge \neg(x \wedge y) \leq \bot$.
5. Follows from $x \wedge 1 \leq x$.
6. Using the evaluation morphism, we know that $x \wedge (x \Rightarrow y) \leq y \leq x \wedge y$.
7. Using the definition of implication $y = y \wedge y \leq y \wedge (x \Rightarrow y)$.
8. The exponential $x \Rightarrow -$ is a right adjoint and it preserves products.

□

Proposition 17 (Complements are negations in Heyting algebras)**.** *If an element has a complement on a Heyting algebra, it must be $\neg x$.*

Proof. Let a a complement of x. By definition, $x \wedge a = \bot$ and therefore $a \leq \neg x$. The reverse inequality can be proven using the lattice properties as

$$\neg x = \neg x \wedge (x \vee a) = \neg x \wedge a.$$

□

Proposition 18 (Characterization of Boolean algebras)**.** *A Heyting algebra is Boolean if and only if $\neg\neg x = x$ for every x; and if and only if $x \vee \neg x = 1$ for every x.*

Proof. In a Boolean algebra the complement is unique and $\neg\neg x = x$. Now, if $\neg\neg y = y$ for every y,

$$x \vee \neg x = \neg\neg(x \vee \neg x) = \neg(\neg x \wedge \neg\neg x) = \top;$$

and then, as $x \vee \neg x = \bot$, $\neg x$ must be the complement of x. We have used the fact that $\neg(x \vee y) = (\neg x) \wedge (\neg y)$ in any Heyting algebra. □

18.3 QUANTIFIERS AS ADJOINTS

Definition 75. Given a relation between sets $S \subseteq X \times Y$, the functors $\forall_p, \exists_p \colon \mathcal{P}(X \times Y) \to \mathcal{P}(Y)$ are defined as

- $\forall_p S = \{y \mid \forall x : \langle x, y\rangle \in S\}$, and
- $\exists_p S = \{y \mid \exists x : \langle x, y\rangle \in S\}$.

Theorem 7. *The functors $\exists_p, \forall_p$ are the left and right adjoints to the inverse image of the projection functor, $p^* : \mathcal{P}(Y) \to \mathcal{P}(X \times Y)$.*

Proof. □

Theorem 8. *Given any function on sets $f : Z \to Y$, the inverse image functor $f^* : \mathcal{P}Y \to \mathcal{P}Z$ has left and right adjoints, called $\exists_f$ and $\forall_f$.*

19

CARTESIAN CLOSED CATEGORIES

19.1 CARTESIAN CATEGORY

Definition 76 (Cartesian category)**.** A **cartesian category** is a category with all finite products.

Definition 77 (Cartesian closed category)**.** A **cartesian closed category** is a category with all finite products and exponentials.

The definition of cartesian closed category can be written in terms of existence of adjoints.

Proposition 19 (Cartesian closed categories and adjoints)**.** *Any category $\mathcal{C}$ is cartesian closed if and only if there exist right adjoints for the following functors*

- $! \colon \mathcal{C} \to 1$*, the unique functor to the terminal category;*
- $\Delta \colon \mathcal{C} \to \mathcal{C} \times \mathcal{C}$*, the diagonal functor;*
- $(- \times A) \colon \mathcal{C} \to \mathcal{C}$*, the product functor, for each $A \in \mathcal{C}$.*

Proof. □

19.2 FRAMES

Proposition 20 (Completeness for posets)**.** *Complete posets are cocomplete. Cocomplete posets are complete.*

Proof. □

Definition 78 (Frames)**. Frames** are complete cartesian closed posets.

20

TOPOI

20.1 SUBOBJECT CLASSIFIER

Definition 79 (Subobject classifier)**.** A **subobject classifier** is an object Ω with a monomorphism $\text{true}\colon 1 \to \Omega$ such that, for every monomorphism $S \to X$, there exists a unique ϕ such that

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

forms a pullback square.

20.2 DEFINITION OF A TOPOS

Definition 80 (Topos)**.** An **elementary topos** (plural *topoi*) is a cartesian closed category with all finite limits and a subobject classifier.

Part V

TYPE THEORY

21

INTUITIONISTIC LOGIC

21.1 CONSTRUCTIVE MATHEMATICS

21.2 AGDA EXAMPLE

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

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

- Machine proof

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

data Nat : Set where
 S : Nat → Nat

data ⊥ : Set where

¬ : (A : Set) → Set
¬ A = (A → ⊥)

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

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

Part VI

CONCLUSIONS

Part VII

APPENDICES

BIBLIOGRAPHY

[Bar84] H.P. Barendregt. *The lambda calculus: its syntax and semantics*. Studies in logic and the foundations of mathematics. North-Holland, 1984.

[Bar92] H. P. Barendregt. Handbook of logic in computer science (vol. 2). chapter Lambda Calculi with Types, pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992.

[Bar94] Henk Barendregt Erik Barendsen. Introduction to lambda calculus, 1994.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic, Volume I*. North-Holland, 1958. Second printing 1968.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[Cur34] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):584–590, 1934.

[dB72] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381 – 392, 1972.

[EM42] Samuel Eilenberg and Saunders MacLane. Group extensions and homology. *Annals of Mathematics*, 43(4):757–831, 1942.

[HHJW07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: Being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, page 1. Microsoft Research, April 2007.

[HM96] Graham Hutton and Erik Meijer. Monadic parser combinators, 1996.

[HM98] Graham Hutton and Erik Meijer. Monadic parsing in haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[HS08] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators: An Introduction*. Cambridge University Press, New York, NY, USA, 2 edition, 2008.

[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.

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

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

[MM94] I. Moerdijk and S. MacLane. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Universitext. Springer New York, 1994.

[O'S16] Bryan O'Sullivan. The attoparsec package. http://hackage.haskell.org/package/attoparsec, 2007–2016.

[P+03] Simon Peyton-Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. http://www.haskell.org/definition/.

[Sel13] Peter Selinger. Lecture notes on the lambda calculus. Expository course notes, 120 pages. Available from 0804.3434, 2013.

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

[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.