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. It should not be written until the end.

Part I

CATEGORY THEORY

1

CATEGORIES

1.1 DEFINITION OF CATEGORY

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

- $\mathcal{C}_0$, a collection[1] whose elements are called **objets**, 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 $\operatorname{dom}(f) \in \mathcal{C}_0$, and a **codomain**, written as $\operatorname{cod}(f) \in \mathcal{C}_0$; a common notation for such morphism is

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

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

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

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.

Part II

LAMBDA CALCULUS

2 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].

2.1 DEFINITION

Definition 2 (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

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

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 $\lambda x, y.M$.

2.2 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 3 (Free variables)**.** The **set of free variables** of a term M is defined inductively as

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

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

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

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.

2.3 α-EQUIVALENCE

Definition 5. **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.

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

2.4 β-REDUCTION

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

Definition 6. 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 $MN \to_\beta MN'$;
- $M \to_\beta M'$ 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 $=$.

2.5 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 7. 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 $\twoheadrightarrow_{\beta\eta}$.

2.6 CONFLUENCE

Definition 8. 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

$$\begin{tikzcd}
& M \arrow[ld, two heads] \arrow[rd, two heads] & \\
N \arrow[rd, dashed, two heads] & & P \arrow[ld, dashed, two heads] \\
& Z &
\end{tikzcd}
\quad
\begin{tikzcd}
& M \arrow[ld] \arrow[rd] & \\
N \arrow[rd, dashed, two heads] & & P \arrow[ld, dashed, two heads] \\
& Z &
\end{tikzcd}
\quad
\begin{tikzcd}
& M \arrow[ld] \arrow[rd] & \\
N \arrow[rd, dashed] & & P \arrow[ld, dashed] \\
& Z &
\end{tikzcd}$$

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.

2.7 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 9 (Parallel one-step reduction)**.** We define the **parallel one-step reduction** relation, $\rhd$ as the smallest relation satisfying that, assuming $P \rhd P'$ and $N \rhd N'$, the following properties of

- reflexivity, $x \rhd x$;

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

hold.

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

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

1. *if $M \rightarrow_{\beta\eta} M'$, 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 $M \rightarrow M'$ are

   - $(\lambda x.M)N \rightarrow M[N/x]$; where we know that, by parallel substitution and reflexivity $(\lambda x.M)N \rhd M[N/x]$.
   - $MN \rightarrow M'N$ and $NM \rightarrow NM'$; 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.
   - $\lambda x.Px \rightarrow P$, 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, $(\lambda x.P)N \twoheadrightarrow (\lambda x.P')N \twoheadrightarrow (\lambda x.P')N' \rightarrow P'[N'/x]$.
   - extensionality, $\lambda x.Px \rhd P'$, where by induction $P \twoheadrightarrow P'$, and trivially, $\lambda x.Px \twoheadrightarrow \lambda x.P'x$.

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

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

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

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

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

Definition 10 (Maximal parallel one-step reduct). The **maximal parallel one-step reduct** M^* of a λ-term M is defined inductively as

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

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

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

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

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

$\square$

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

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

2.8 NORMALIZATION

Definition 11 (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 12. 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

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

3

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.

3.1 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 13 (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.

3.2 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 14 (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 15 (Typing context)**.** A **typing context** is a sequence of typing assumptions $x_1 : A_1, \dots, x_n : A_n$, 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 16. 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.

3.3 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 17 (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 18 (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 19 (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, $\overline{\sigma}\pi = \tau$.

3.4 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 20 (Most general unifier)**.** A substitution ψ is called an **unifier** of two sequences of type templates $\{A_i\}_{i=1,\dots,n}$ and $\{B_i\}_{i=1,\dots,n}$ if $\overline{\psi}A_i = \overline{\psi}B_i$ for any i. We say that it is the **most general unifier** if given any other unifier ϕ exists a substitution φ such that $\phi = \overline{\varphi} \circ \psi$.

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

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

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

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

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

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

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

Using unification, we can define type inference.

Theorem 2 (Type inference). *The algorithm* `typeinfer`(M, B)*, defined as follows, finds the most general substitution* σ *such that* $x_1 : \sigma A_1, \dots, x_n : \sigma A_n \vdash M : \overline{\sigma} B$ *is a valid typing judgment if it exists; and fails otherwise.*

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

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

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

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

Corollary 1 (Principal type property). *Every typeable pure* λ*-term has a principal type.*

Proof. Given a typeable term M, we can compute `typeinfer`$(x_1 : A_1, \dots, x_n : A_n \vdash M : B)$, where $x_1, \dots, x_n$ are the free variables on M and $A_1, \dots, A_n, B$ are fresh type variables. By virtue of Theorem 2, the result is the most general type of M if we assume the variables to have the given types. □

4

THE CURRY-HOWARD CORRESPONDENCE

4.1 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 21 (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 22 (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.

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

   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, $(\text{case } m \text{ of } [a].P[\text{inl } a/c];\ [b].P[\text{inr } b/c]) \to_\eta P[m/c]$.

4.2 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.

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

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.

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

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

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\overset{1}{\vdots}\\ A\end{array} \quad \begin{array}{c}\overset{2}{\vdots}\\ B\end{array}}{A \wedge B}}{A}}{\vdots} \qquad \Longrightarrow \qquad \begin{array}{c}\overset{1}{\vdots}\\ \dfrac{A}{\vdots}\end{array}$$

and, similarly, on the other side as

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

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

$$\begin{array}{c}\dfrac{\dfrac{\begin{array}{c}1\\ \vdots\\ A\end{array}}{A+B}\quad\begin{array}{c}[A]\\ 2\\ \vdots\\ C\end{array}\quad\begin{array}{c}[B]\\ 3\\ \vdots\\ C\end{array}}{C}\\ \vdots\end{array} \quad\Longrightarrow\quad \begin{array}{c}1\\ \vdots\\ A\\ 2\\ \vdots\\ \underline{C}\\ \vdots\end{array}$$

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

$$\begin{array}{c}\dfrac{\dfrac{\begin{array}{c}1\\ \vdots\\ B\end{array}}{A+B}\quad\begin{array}{c}[A]\\ 2\\ \vdots\\ C\end{array}\quad\begin{array}{c}[B]\\ 3\\ \vdots\\ C\end{array}}{C}\\ \vdots\end{array} \quad\Longrightarrow\quad \begin{array}{c}1\\ \vdots\\ B\\ 3\\ \vdots\\ \underline{C}\\ \vdots\end{array}$$

4.3 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 1 (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

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

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

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

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

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

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
```

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.

5

LAMBDA EXPRESSIONS

5.1 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, *De Bruijn indexes* and *De Bruijn notation* are different concepts and we are going to use only the former.

Definition 23 (De Bruijn indexed terms)**.** We define recursively the set of λ-terms using de Bruijn notation following this BNF

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

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

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

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

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

5.2 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
```

5.3 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..]
```

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

5.4 PRINCIPAL TYPE INFERENCE

6

PARSING

6.1 MONADIC PARSER COMBINATORS

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

The use of monads for parsing 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 monad 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

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

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

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

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

6.2 PARSEC

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

7

USAGE

7.1 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> _
```

7.2 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.

7.3 CODEMIRROR LEXER

8

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.

8.1 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**

$$\mathrm{Hom}(A \times B, C) \cong \mathrm{Hom}(A, \mathrm{Hom}(B, C))$$

or the definition of exponentials.

8.2 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,

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

Or, in general

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

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 $\mathtt{Succ}(\mathtt{Succ}(\mathtt{Zero}))$, 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.

8.3 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
```

8.4 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
```

8.5 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.

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

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

9

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.

9.1 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 2 (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 3 (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.

9.2 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
$\to$	Function type	Functions from a type to another.
$\times$	Product type	Cartesian product of types.
$+$	Union type	Disjoint union of types.
$\top$	Unit type	A type with exactly one element.
$\bot$	Void type	A type with no elements.

And the following typed constructors are added to the language,

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

which correspond to the constructors we described on previous sections. The only new 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 4 (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.

9.3 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 3. *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 $\neg\neg(A \vee \neg A)$. $\square$

Note that this is, in fact, an intuitionistic proof. Although it seems to use the intuitionistically forbidden technique of prooving 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

```
\f.absurd (f (inr (\a.f (inl a))))
---> 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

TYPE THEORY

10 INTUITIONISTIC LOGIC

10.1 CONSTRUCTIVE MATHEMATICS

10.2 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 V

CONCLUSIONS

Part VI

APPENDICES

BIBLIOGRAPHY

[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.

[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.

[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.

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

[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.