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

I CATEGORY THEORY 5
1 CATEGORIES 6
1.1 Definition of category 6
II LAMBDA CALCULUS 7
2 UNTYPED λ-CALCULUS 8
2.1 Definition 8
2.2 Free and bound variables, substitution 8
2.3 α-equivalence 9
2.4 β-reduction 10
2.5 η-reduction 10
2.6 Confluence 11
2.7 The Church-Rosser theorem 11
3 SIMPLY TYPED LAMBDA CALCULUS 15
3.1 Simple types 15
3.2 Raw typed lambda terms 15
3.3 Typing rules for the simply-typed lambda calculus 16
III MIKROKOSMOS 17
4 LAMBDA EXPRESSIONS 18
4.1 De Bruijn indexes 18
4.2 Substitution 19
4.3 De Bruijn-terms and λ-terms 19
5 PARSING 22
5.1 Monadic parser combinators 22
5.2 Parsec 23
6 USAGE 24
6.1 Jupyter kernel 24
6.2 CodeMirror lexer 24
7 PROGRAMMING IN THE UNTYPED λ-CALCULUS 25
7.1 Basic syntax 25
7.2 A technique on inductive data encoding 26
7.3 Booleans 27
7.4 Natural numbers 28
7.5 Lists 29

IV TYPE THEORY 31
8 INTUITIONISTIC LOGIC 32
8.1 Constructive mathematics 32
8.2 The double negation of LEM is provable 32
V CONCLUSIONS 33
VI APPENDICES 34

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. 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 $\mathrm{dom}(f) \in \mathcal{C}_0$, and a **codominio**, 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

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

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. 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

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

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 $\lambda x, y.M$.

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. 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. 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; 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

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

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. $\square$

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

$\square$

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 $\rightarrow_{\beta\eta}$. $\square$

3

SIMPLY TYPED LAMBDA CALCULUS

We will give now a presentation of the **simply-typed lambda calculus** based on [Sel13].

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. We will also assume that a **unit** type, 1 exists; the unit type will have only one inhabitant.
Definition 11. The set of **simple types** is given by the following Backus-Naur form

$$\mathtt{Type} ::= 1 \mid \iota \mid \mathtt{Type} \to \mathtt{Type} \mid \mathtt{Type} \times \mathtt{Type}$$

where 1 is a one-element type and ι is any *basic type*.

That is to say that, for every two types A, B, there exist a **function type** $A \to B$ and a **pair type** $A \times B$.

3.2 RAW TYPED LAMBDA TERMS

We will now define the terms of the typed lambda calculus.
Definition 12. The set of **typed lambda terms** is given by the BNF

$$\mathtt{Term} ::= * \mid x \mid \mathtt{Term}\mathtt{Term} \mid \lambda x^{\mathtt{Type}}.\mathtt{Term} \mid \langle \mathtt{Term}, \mathtt{Term} \rangle \mid \pi_1 \mathtt{Term} \mid \pi_2 \mathtt{Term}$$

Besides the previously considered term application and a special element $*$ which will be the unique inhabitant of the type 1; we now introduce a typed lambda abstraction and an explicit construction of the pair element with its projections.

3.3 TYPING RULES FOR THE SIMPLY-TYPED LAMBDA CALCULUS

The set of raw typed lambda 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 13. 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 type of $*$ is 1, the rule $(*)$ builds this element.

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

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

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

3. The $(pair)$ rule allow us to build pairs by their components. It acts as a constructor of pairs.

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

4. The (π_1) and (π_2) rules give the semantics of a product with two projections to the pair terms. If we have a pair $m : A \times B$, then $\pi_1 m : A$ and $\pi_2 m : B$. They act as two different destructors of pairs.

$$(\pi_1)\ \frac{\Gamma \vdash m : A \times B}{\Gamma \vdash \pi_1 m : A} \qquad (\pi_2)\ \frac{\Gamma \vdash m : A \times B}{\Gamma \vdash \pi_2 m : B}$$

5. The (abs) introduces a well-typed lambda abstraction. If we have a $h : B$ term depending on $x : A$, we can create a lambda abstraction from this term. It acts as a constructor of function terms.

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

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

Part III

MIKROKOSMOS

We have developed **Mikrokosmos**, a lambda calculus interpreter written in the purely functional programming language Haskell [HHJW07]. It aims to provide students with a tool to learn and understand lambda calculus.

4

LAMBDA EXPRESSIONS

4.1 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 15 (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`:

```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.

4.2 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
```

4.3 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.

```haskell
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

```haskell
tobruijn :: Map.Map String Integer -- ^ names of the variables used
         -> Context                -- ^ names already binded on the scope
         -> NamedLambda            -- ^ initial expression
         -> Exp
-- Every lambda abstraction is inserted in the variable dictionary,
-- and every number in the dictionary increases to reflect we are entering
-- 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.

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

-- | A function translating a deBruijn expression into a
-- natural lambda expression.
nameIndexes :: [String] -> [String] -> Exp -> NamedLambda
nameIndexes _    _   (Var 0) = LambdaVariable "undefined"
nameIndexes used _   (Var n) = LambdaVariable (used !! pred (fromInteger n))
```

```
nameIndexes used new (Lambda e) =
  LambdaAbstraction (head new) (nameIndexes (head new:used) (tail new) e)
nameIndexes used new (App f g) =
  LambdaApplication (nameIndexes used new f) (nameIndexes used new g)
```

5

PARSING

5.1 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.

5.2 PARSEC

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

6

USAGE

6.1 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.

6.2 CODEMIRROR LEXER

7

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.

7.1 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
--- => 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.

7.2 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.

7.3 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
```

7.4 NATURAL NUMBERS

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

- the 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 the 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
```

7.5 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.

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

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
```

Part IV

TYPE THEORY

8

INTUITIONISTIC LOGIC

8.1 CONSTRUCTIVE MATHEMATICS

8.2 THE DOUBLE NEGATION OF LEM IS PROVABLE

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 firstly are going to prove 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 $\neg\neg(A \vee \neg A)$.
- Machine proof

```
id : {A : Set} → A → A
id a = 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.

[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.