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. 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 are going to use the term *collection* to denote some unspecified formal notion of collection that could be given by sets or proper classes.

Part II

LAMBDA CALCULUS

2

TYPED LAMBDA CALCULUS

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

2.1 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 2. The set of **simple types** is given by the following Backus-Naur form

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

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** $A \times B$.

2.2 RAW TYPED LAMBDA TERMS

We will now define the terms of the typed lambda calculus.

Definition 3. The set of **typed lambda terms** is given by the BNF

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

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.

2.3 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 4. A **typing context** is a sequence of typing assumptions $x_1 : A_1, \ldots, x_n : A_n$, where no variable appears more than once.

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

1. The 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 5. 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.

3

PARSING

3.1 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

```
1 newtype Parser a = Parser (String -> [(a,String)])
2
3 parse :: Parser a -> String -> [(a,String)]
4 parse (Parser p) = p
5
6 instance Monad Parser where
7   return x = Parser (\s -> [(x,s)])
8   p >>= q  = Parser (\s ->
9               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

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

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

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

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

3.2 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.

4

USAGE

4.1 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.

4.2 CODEMIRROR LEXER

Part IV

CONCLUSIONS

BIBLIOGRAPHY

[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.

[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. *Dalhousie University*, 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.