Acknowledgments

Thanks your peoples here.

Statement of Integrity

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Resumo

Abstract em português.

Palavras-chave 3 a 5 palavras-chave, ordenadas alfabeticamente e separadas por vírgulas

Abstract

Your abstract here.

Keywords 3-5 keywords alphabetically ordered and comma-separated.

"We adore chaos because we love to produce order."

M. C. Escher

Contents

Chapter 1

Introduction

1.1 Motivation

There is no motivation, yet we need to write one.

1.2 Objectives

Formalise results about λ-calculus variants in *Coq*.

1.3 Document Structure

List your chapters here, with a very brief description of each one.

Chapter 2

Background

In this chapter, we intend to take the example of formalising some theory about the λ-calculus as a motivation to display some core aspects involved in the more elaborate formalisations ahead.

2.1 Basic Lambda Calculus Theory

2.1.1 Syntax

[3] [1]

Definition 1. *Given an infinite sequence of variables V, we define the set of λ-terms, Λ, inductively as follows:*

1. *(variable)* $x \in V \implies x \in \Lambda$
2. *(application)* $M, N \in \Lambda \implies (MN) \in \Lambda$
3. *(abstraction)* $x \in V, M \in \Lambda \implies (\lambda x.M) \in \Lambda$

Notation.

1. *(variable names) Variables in V will be denoted with the lowercase letters w, x, y, z with or without number-subscripts.*
2. *(λ-term names) Terms in Λ will be denoted with the uppercase letters M, N, P, Q with or without number-subscripts.*
3. *(multiple abstractions) For every term $M \in \Lambda$, the term $\lambda x_1 x_2 \dots x_n.M$ will be syntactic sugar for the term $\lambda x_1.\lambda x_2.\dots.\lambda x_n.M$.*
4. *(application left association) For every terms $M_i \in \Lambda$, the term $M_1 M_2 \dots M_n$ will be syntactic sugar for the term $((M_1 M_2)\dots)M_n$.*

Definition 2 (Free variables)**.** *For every $M \in \Lambda$, we define the set of free variables in M, $FV(M)$, recursively as follows:*

1. $FV(x) = \{x\}$
2. $FV(MN) = FV(M) \cup FV(N)$
3. $FV(\lambda x.M) = FV(M) - \{x\}$

When a variable occurring in a term is not free it is said to be bound.

Definition 3 (α-conversion)**.** *We identify*

Convention. *[1] We will use the same*

Definition 4 (Substitution)**.** *For every*

Definition 5 (Compatible Relation)**.** *We say that a binary relation $R : \Lambda \times \Lambda$ is compatible if:*

1. $(M, N) \in R \implies \forall x \in V.(\lambda x.M,)$

2.1.2 Typed Terms

2.2 Mechanising Meta-theory in Coq

The formalisation we aim at is dependent on the theory provided by the *Coq* proof assistant - the Calculus of Inductive Constructions. We will follow assuming a basic knowledge on *Coq* and its syntax to define inductive types and proof techniques.

2.2.1 Binding Variables

How could we define the syntax of the untyped λ-terms in *Coq*? Probably, we would describe something similar to:

```
Inductive term :
| Var : var
| Abs : var -> term
| App : term -> term -> term
end.
```

We would still have to clarify what we mean by variables (the above var type). Here lies a common and subtle problem that is usually not noticed when describing definitions in a paper. If we want these variables to be the usual reserved strings (say, $x, x_0, x_1, x_2, \ldots, y, y_0, y_1, y_2, \ldots, z, z_0, z_1, z_2, \ldots$) we would fall into some of the known problems [4]:

- α-equivalence: We want to be able to identify which variables are bound in our term but the name they hold is indifferent. A simple example is the term $Abs\ x\ (Var\ x)$ that is syntactically different from the term $Abs\ y\ (Var\ y)$. We desire to compare *terms modulo a change of bound variables* [1].

- variable capture: When dealing with substitutions, we will swap a variable by a new term with possibly free variables that can get bound to an unwanted abstraction. For example, if we were to substitute the free occurrence of $Var\ y$ in the term $Abs\ x\ (Var\ y)$ by $Var\ x$ we would get the term $Abs\ x\ (Var\ x)$, with the variable x now bound.

Furthermore, we notice that having such freedom to choose the names of our bound variables is annoying (and even unnecessary for our case). This motivates our use of de Bruijn syntax when formalising our calculi with binders.

2.2.2 De Bruijn Syntax

In the 1970s, de Bruijn started working on the *AUTOMATH* proof assistant and proposed a simplified syntax to deal with generic binders [2]. This approach is claimed to be good for meta-lingual discussion and for the computer and computer programme. In contrast, this syntax is farther away from the human reader.

The main idea is to treat variables as indices (represented by natural numbers) and to interpret these indices as the distance to the respective binder. In de Bruijn syntax for λ-terms, we only have one representative for the identity terms $\lambda x.x$, $\lambda y.y$, $\lambda z.z$ ..., that is, $\lambda.0$. This way, we do not decide the names for the bound variables in a term - we are obliged to use the specific indices that bind a variable to a certain binder.

Let us see how using this approach would change our first definition of the untyped λ-terms in *Coq*.

```
Inductive term :
| Var : nat
| Abs : term
| App : term -> term -> term
end.
```

Now, we do not have to specify the variable name we want to bind - this was unnecessary! Using this definition, we also eliminate our problem of identifying terms modulo α-equivalence.

Variable capture can also be avoided, but this requires some care when defining the substitution operations. ~~The next question to answer is: What are these substitutions and how do we instantiate them in our terms?~~

2.2.3 Explicit Substitutions and the Sigma Calculus

2.2.4 The Autosubst Library

The *Autosubst* library for *Coq* is one of the many solutions to generically deal with binders using de Bruijn syntax and to support substitution operations in an inductively defined calculi. This library also supported some automation to define the substitution operations for simple inductive definitions of terms and to prove some results that deal with elaborate substitution equalities (that take advantage of the substitution lemmas).

Now, the changes

```
Inductive term :
```

```
| Var : var
| Abs : { bind term }
| App : term -> term -> term
end.
```

Chapter 3

The Multiary Lambda Calculus and Its Canonical Fragment

3.1 The Multiary Lambda Calculus (λm)

3.1.1 A Nested Inductive Type

3.1.2 A Sequent Calculus Fragment

3.2 The Canonical Fragment ($\vec{\lambda} m$)

3.2.1 Formalising a Subsystem

3.2.2

3.2.3 Conservativeness

Chapter 4

The Isomorphic Fragment of Lambda Calculus in λm

Chapter 5

Discussion

Chapter 6

Conclusions

Final chapter, present your conclusions.

6.1 Summary of Findings

Highlight per-chapter content here, with a general conclusion in the end.

6.2 Future Work

There's no lack of future work.

Appendix A

Example

This is what an Appendix looks like.

Bibliography

[1] H. P. Barendregt. *The lambda calculus*. Studies in Logic and the Foundations of Mathematics. Elsevier Science, London, England, 2 edition, Oct. 1987.

[2] N. 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. ISSN 1385-7258. doi: https://doi.org/10.1016/1385-7258(72)90034-0. URL https://www.sciencedirect.com/science/article/pii/1385725872900340.

[3] J. R. Hindley. *Basic Simple Type Theory*. Cambridge University Press, Cambridge, July 1997.

[4] T. Winterhalter. *Formalisation and meta-theory of type theory*. PhD thesis, Université de Nantes, 2020.