Universidade do Minho
Escola de Ciências

Miguel José de Melo Alves

On the mechanisation of the multiary lambda calculus and subsystems

Dissertação de Mestrado
Mestrado em Matemática e Computação
Área de especialização de Computação

Trabalho efetuado sob a orientação dos
Professor Doutor Luís Filipe Ribeiro Pinto
Professor Doutor José Carlos Soares do Espírito Santo

Setembro de 2025

Acknowledgments

I feel blessed to write this, as I contemplate so many people that I have to thank, as I see so many people that I love and love me back.

First and foremost, I must thank my family for allowing me to pursue what I love and for unconditionally supporting my passions.

Second, I want to deeply thank my girlfriend Inês for being my companion throughout this year of confusion, tiredness and grumpiness. I still owe you a complete explanation of what λ-calculus is.

Third, I want to thank all my friends...should I name them? To Nando, Marco and David, as you are a second kind of girlfriend to me. To Gaspar, Rodrigo and Kiko, as you have known me for a long time and part of who I am today is because of you. To Pepa, as you may not understand how much of a help you were to me throughout this year. To Gui, Zé, Bruno, Sofia, Manada, Gonçalo, Jason, Jonny and to every algebra class that made us closer. To Faustino, Ricardo, Guilherme and Filipa, for making me feel more at home in Braga. To my long time friends, Couves e Beterrabas, for being with me in such an important and playful time in my life. To my CVX group, VitaminaC, for walking with me. To CREU, Missão País, Fonte da Prata and to every friend that I have made in those special places - a piece of my heart lies with each of you.

Fourth and last, I want to thank everyone who I feel had a part in my logical education, from Rola and Rita to professor Filipe and professora Alice Viveiros. And, of course, to my kind supervisors. Thank you, Professor Luís, for your patience and for your focused, optimistic and close help throughout this joyful process. Thank you, Professor José Carlos, for your enthusiasm and for every exciting conversation we have shared.

To everyone that casually asked what I was studying in my dissertation and kept their attention at my explanation - because attention without distraction is a prayer - I thank you.

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

Looking at the title for this dissertation, we can explain what motivated the following dissertation by asking ourselves: “Why mechanise?”, “Why metatheory?” and “Why the multiary λ-calculus?”.

Before addressing these questions formally, we could just say that mechanising mathematics is an enjoyable task. And that could be all we say about our motivation. Even if our work in mathematics had no application or direct consequences, the fun of mechanising it in a proof assistant would be a good enough motivation. Mechanising mathematics is like a computer game for a mathematician.

Why mechanise? By mechanisation we mean a well-founded description of a mathematical object using a proof assistant. Such proof assistants have attracted the attention of mathematicians because of the reliability and automation they provide for writing computer verified proofs [16]. There has also been an increasing interest by engineers in the use of such tools for the security guarantees achieved when formally proving properties about computer programmes [22].

One could even argue that any work of mechanisation is useful, because it will:

1. result in a computer-verified work,
2. expose the difficulties behind any mathematical formalisation,
3. provide automation for routine and tedious parts,
4. potentially allow some theory to be extended with less cost.

Some of the above mentioned items may even be highlighted when the mechanisation refers to metatheory.

Why metatheory? It is often argued that metatheoretical proofs *“are long, contain few essential insights, and have a lot of tedious but error-prone cases”* [26]. This provides fertile ground for computer verification and automation of proofs. Furthermore, the mechanisation of metatheory has also gained some attention in the past 20 years [5, 2].

In our case, mechanising theoretical results related to the multiary λ-calculus could enable new ways of continuing the work being done in this topic. Curiously, our work with an unusual version of the λ-calculus could even suggest some improvements in already mature tools for mechanising metatheory (as is the case with the used *Autosusbt* library for the *Rocq Prover*).

Why the multiary λ-calculus? In the beginning of [28, Chapter 7.3], one is confronted with a natural question: *"Natural deduction proofs correspond to λ-terms with types, and Hilbert style proofs correspond to combinators with types. What do sequent calculus proofs correspond to?"*. This question has its starting point in the well-known Curry-Howard isomorphism, that relates natural deduction proofs with λ-terms with types, as said above.

Many (naive) alternatives are given in the aforementioned book, but none that can match the process of cut-elimination with normalisation. In the novel paper of Herbelin [17], a multiary version of the λ-calculus (with explicit substitutions) called $\overline{\lambda}$ is introduced, whose typing rules correspond to a fragment of the sequent calculus and reduction rules behave as cut elimination.

We are interested in the study of a slightly different version of $\overline{\lambda}$ which has no explicit substitutions [13, 14], here named $\boldsymbol{\lambda m}$. Studying the computational meaning behind the sequent calculus is one of the main motivations for considering such systems, as they provide meaningful extensions for the ordinary λ-calculus.

1.2 Objectives and contributions

The theoretical objectives for this dissertation are the study of:

1. system $\boldsymbol{\lambda m}$, reduction rules, typing rules and standard results like subject reduction;
2. the canonical subsystem of $\boldsymbol{\lambda m}$;
3. the conservativeness of the canonical subsystem over $\boldsymbol{\lambda m}$;
4. the isomorphism between the canonical subsystem and $\boldsymbol{\lambda}$.

We say theoretical objectives because the complete objective is to mechanise in the *Rocq Prover* each of the mentioned items. The practical (in the sense of the mechanisation task) objectives of this dissertation are first to understand the proof assistant in order to fully develop a mechanisation of the definitions and proofs that were studied using pen-and-paper. Concretely, we have the objective of understanding how one can use the *Rocq Prover* to define systems that deal with variable binding, to define subsystems, to define typing rules, to prove isomorphisms and so on.

A last (and challenging) objective related to the mechanisation is to formalise every definition and proof as close as possible to the pen-and-paper versions, assuring clean and simple presentations.

This dissertation presents a mechanised version of the system $\boldsymbol{\lambda m}$ and its associated metatheory - including an isomorphism of its subsystem with the simply typed λ-calculus and the conservativeness results - within the *Rocq Prover*. We know of no other works formalising this metatheory. This formalised body of work provides a computer-verified and highly accessible foundation for future study and can be found in an open-source *GitHub* repository [1].

1.3 Document structure

This dissertation is organised as follows:

chapter 2 serves as an introduction to the ordinary $\boldsymbol{\lambda}$-calculus, also containing a second style of $\boldsymbol{\lambda}$-calculus presentation, without variable names (also called de Bruijn representation). This chapter also includes a mechanisation of this system as a way to introduce many proof-assistant concepts used in further chapters.

chapter 3 introduces the system $\boldsymbol{\lambda m}$ and its canonical subsystem, along with some simple results. It also includes a last chapter that provides a walk-through of the mechanised definitions and proofs.

chapter 4 independently introduces a new system called $\vec{\boldsymbol{\lambda}}$, that is isomorphic to the introduced canonical subsystem of $\boldsymbol{\lambda m}$. This system will help clarify the mechanisation of the result of conservativeness that can be found in this chapter. The last section also provides a mechanisation overview, similar to the previous chapter.

chapter 5 is only about the isomorphism between the ordinary $\boldsymbol{\lambda}$-calculus and system $\vec{\boldsymbol{\lambda}}$.

chapter 6 lists our contributions, discusses our approaches and related work and then points towards possible future work.

Every *Rocq Prover* module/script referred throughout this document may be found in the previously mentioned *GitHub* repository.

[1] https://github.com/thetruezau/LambdaM

Chapter 2

Background

This chapter introduces essential background for the reading of this dissertation. First, we introduce the well-known simply typed **λ**-calculus. Then, we delve into a known variation of the introduced **λ**-calculus theory using de Bruijn indices, that has known facilities when it comes to mechanisation. Lastly, we present and explain a mechanisation of the simply typed **λ**-calculus in the *Rocq Prover*.

2.1 Simply typed λ-calculus

For the basic concepts and basic theory of the untyped **λ**-calculus we refer to [6]. For what types and the simply typed lambda calculus is about we refer to [7] and [18].

2.1.1 Syntax

Definition 1 (**λ**-terms)**.** *The* **λ***-terms are defined by the following grammar:*

$$M, N \ ::= \ x \mid (\lambda x.M) \mid (MN),$$

where x denotes a variable.

Remark.

1. *A denumerable set of variables is assumed and letters x, y, z range over this set.*

2. *An abstraction is a* **λ***-term of the kind $(\lambda x.M)$, that will bind occurrences of x in the term M (also called scope of the abstraction), much like a function $x \mapsto M$.*

3. *An application is a* **λ***-term of the kind $(M_1 M_2)$, where M_1 has the role of function and M_2 has the role of argument.*

Notation. *We shall assume the usual notational conventions on* **λ***-terms:*

1. *Outermost parentheses are omitted.*

2. *Multiple abstractions can be abbreviated as $\lambda xyz.M$ instead of $\lambda x.(\lambda y.(\lambda z.M))$.*

3. *Multiple applications can be abbreviated as MN_1N_2 instead of $(MN_1)N_2$.*

Now we may define some syntactic notions.

Definition 2 (Bound/free occurrence)**.** *We say that the variable x occurs bound when it occurs in the scope of an abstraction λx and say that it occurs free otherwise.*

As an illustration of the previous concept, consider the term $M = x(\lambda x.x)$. The variable x occurs both free and bound in this term.

We can easily calculate the set of free variables for a given term.

Definition 3 (Free variables)**.** *For every* $\boldsymbol{\lambda}$*-term* M*, we recursively define the set of free variables in* M*,* $FV(M)$*, as follows:*

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

Now, we will consider a sequence of steps taken from [6], in order to define a substitution operation that avoids the capture of free variables. This is, a substitution operation that does not swap free variables for bound variables.

Definition 4 (Renaming of bound variables)**.** *A renaming of bound variables in a* $\boldsymbol{\lambda}$*-term* M *is the replacement of a part* $\lambda x.N$ *of* M *by* $\lambda y.N\langle x := y\rangle$*, where* y *does not occur at all in* N *and* $N\langle x := y\rangle$ *denotes the naive substitution operation.*

Observe that $N\langle x := y\rangle$ *is capture-avoiding as* y *is carefully chosen to not occur in* N*.*

Definition 5 (α-congruence)**.** *Given* $\boldsymbol{\lambda}$*-terms* M, N*, we say that* M *is* α*-congruent with* N*, namely* $M \equiv_\alpha N$*, when they are equal up to a series of renamings of bound variables.*

As an example, we can see that $\lambda x.\lambda y.x \equiv_\alpha \lambda z.\lambda y.z \equiv_\alpha \lambda z.\lambda x.z \equiv_\alpha \lambda y.\lambda x.y$.

Given this notion, we will prefer to identify α-congruent $\boldsymbol{\lambda}$-terms. Moreover, we are now able to introduce the variable convention that is proposed in [6].

Convention. *If some* $\boldsymbol{\lambda}$*-terms* $M, M', \ldots$ *occur in a certain mathematical context (of a definition, or proof, etc...), then all bound variables in these terms are chosen to be different from the free variables.*

Definition 6 (Substitution)**.** *For every* $\boldsymbol{\lambda}$*-term* M*, we recursively define the substitution of* N *for the free occurrences of* x *in* M*,* $M[x := N]$*, as follows:*

$$
\begin{aligned}
&x[x := N] = N;\\
&y[x := N] = y, \text{ with } x \neq y;\\
&(\lambda y.M_1)[x := N] = \lambda y.(M_1[x := N]);\\
&(M_1M_2)[x := N] = (M_1[x := N])(M_2[x := N]).
\end{aligned}
$$

In the third equation, there is no need to say that $y \neq x$ *and that* $y \notin FV(N)$ *as this is the case by the variable convention.*

At last, we introduce some standard notions related to the β-reduction.

Definition 7 (Compatible Relation)**.** *Let R be a binary relation on $\boldsymbol{\lambda}$-terms. We say that R is compatible if it satisfies:*

$$\frac{(M_1, M_2) \in R}{(\lambda x.M_1, \lambda x.M_2) \in R} \qquad \frac{(M_1, M_2) \in R}{(NM_1, NM_2) \in R} \qquad \frac{(M_1, M_2) \in R}{(M_1N, M_2N) \in R}$$

Notation. *Given a binary relation R on $\boldsymbol{\lambda}$-terms, we define:*

$\to_R$ *as the compatible closure of* R;

$\twoheadrightarrow_R$ *as the reflexive and transitive closure of* $\to_R$.

Definition 8 (β-reduction)**.** *Consider the following binary relation on $\boldsymbol{\lambda}$-terms:*

$$\beta = \{((\lambda x.M)N, M[x := N]) \mid \text{for every variable } x \text{ and } \boldsymbol{\lambda}\text{-terms } M, N\}.$$

We call one step β-reduction to the relation $\to_\beta$ and multistep β-reduction to the relation $\twoheadrightarrow_\beta$.

Definition 9 (β-normal form)**.** *We say that a $\boldsymbol{\lambda}$-term t is in β-normal form (or irreducible by $\to_\beta$) when there exists no $\boldsymbol{\lambda}$-term t' such that*

$$t \to_\beta t'.$$

Definition 10. *We inductively define the sets of $\boldsymbol{\lambda}$-terms NF and NA as follows:*

$$\frac{}{x \in \mathit{NA}} \qquad \frac{M_1 \in \mathit{NA} \qquad M_2 \in \mathit{NF}}{M_1M_2 \in \mathit{NA}} \qquad \frac{M \in \mathit{NA}}{M \in \mathit{NF}} \qquad \frac{M \in \mathit{NF}}{\lambda x.M \in \mathit{NF}}$$

Claim 1. *Given a $\boldsymbol{\lambda}$-term M, the following are equivalent:*

(i) $M \in$ NF.

(ii) M is in β-normal form.

We leave this claim here, but we will show the mechanised proof for $\boxed{\text{(i)} \Rightarrow \text{(ii)}}$ in the last section of this chapter. The proof for $\boxed{\text{(ii)} \Rightarrow \text{(i)}}$ is also mechanised in the script repository of our development.

2.1.2 Types

Definition 11 (Simple Types)**.** *The simple types are defined by the following grammar:*

$$A, B, C ::= p \mid (A \supset B),$$

where p denotes a type variable.

Remark.

1. *A denumerable set of atomic variables is assumed and letters p, q, r range over this set.*

2. *Notice that we use the symbol $\supset$, coming from logic, to denote implication. This is motivated by the well-known correspondence between function types and implicational proposition, through the Curry-Howard isomrphism.*

Notation. *We will assume the usual notational conventions on simple types.*

1. *Outermost parentheses are omitted.*

2. *Types associate to the right. Therefore, the type $A \supset (B \supset C)$ may often be written simply as $A \supset B \supset C$.*

Definition 12 (Type-assignment)**.** *A type-assignment $M : A$ is a pair of a **λ**-term and a simple type. We call subject to the **λ**-term M and predicate to the simple type A.*

Definition 13 (Context)**.** *A context $\Gamma, \Delta, \ldots$ is a finite (possibly empty) set of type-assignments whose subjects are variables of **λ**-terms and which is consistent. By consistent we mean that no variable is the subject of more than one type-assignment.*

Notation. *We may simplify the set notation of contexts as follows:*

$$\begin{array}{rcl} x : A, \ldots, y : B & \textit{for} & \{x : A, \ldots, y : B\} \\ x : A, \ldots, y : B, \Gamma & \textit{for} & \{x : A, \ldots, y : B\} \cup \Gamma. \end{array}$$

Definition 14 (Sequent)**.** *A sequent $\Gamma \vdash M : A$ is a triple of a context, a **λ**-term and a simple type.*

Definition 15 (Typing rules for **λ**-terms)**.** *The following typing rules inductively define the notion of derivable sequent.*

$$\frac{}{x : A, \Gamma \vdash x : A}\ \textit{Var} \qquad \frac{x : A, \Gamma \vdash M : B}{\Gamma \vdash \lambda x.M : A \supset B}\ \textit{Abs} \qquad \frac{\Gamma \vdash M : A \supset B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}\ \textit{App}$$

A sequent is derivable when it is at the root of a tree constructed by the successive application of the typing rules and whose leaves are instances of the Var-rule.

2.2 λ-calculus with de Bruijn syntax

In the 1970s, de Bruijn started working on the *Automath* proof checker and proposed a simplified syntax to deal with generic binders [9]. This approach is claimed by the author to be good for meta-lingual discussion and for implementation in computer programmes. In contrast, this syntax is further away from the human reader. This section will serve as an intermediate step to the mechanised version of the simply typed **λ**-calculus described in the next section.

The main idea behind de Bruijn syntax (or sometimes called de Bruijn indices) is to treat variables as natural numbers (or indices) and to interpret these numbers as the distance to the respective binder. Therefore, we will call these terms *nameless*.

Definition 16 (Nameless **λ**-terms)**.** *The nameless **λ**-terms are defined by the following grammar:*

$$M, N \ ::= \ i \mid \lambda.M \mid MN \ ,$$

where i ranges over the natural numbers.

Remark. *Nameless **λ**-terms have no α-conversion since there is no freedom to choose the names of bound variables.*

We show below some examples that illustrate the connection of ordinary and nameless syntax for **λ**-terms.

$$\lambda x.x \rightsquigarrow \lambda.0$$
$$\lambda x.\lambda y.x \rightsquigarrow \lambda.\lambda.0$$
$$\lambda x.\lambda y.x \rightsquigarrow \lambda.\lambda.1$$

Now, we will present a different formulation for the concept of substitution, adequate to deal with nameless **λ**-terms.

Definition 17 (Substitution)**.** *A substitution $\sigma, \tau, \ldots$ over nameless **λ**-terms is a function mapping natural numbers (indices) to nameless **λ**-terms.*

Here are some examples of useful substitutions.

$$id(k) = k$$
$$\uparrow(k) = k + 1$$
$$(M \cdot \sigma)(k) = \begin{cases} M & \text{if } k = 0 \\ \sigma(k-1) & \text{if } k > 0 \end{cases}$$

Definition 18 (Instantiation and composition)**.** *The operation of instantiating a nameless **λ**-term M*

under a substitution σ, $M[\sigma]$, is recursively defined by the following equations:

$$
\begin{aligned}
i[\sigma] &= \sigma(i);\\
(\lambda.M_1)[\sigma] &= \lambda.(M_1[0 \cdot (\uparrow \circ \sigma)]);\\
(M_1M_2)[\sigma] &= (M_1[\sigma])(M_2[\sigma]);
\end{aligned}
$$

where the composition of two substitutions is mutually defined as $(\tau \circ \sigma)(k) = \sigma(k)[\tau]$.

This definition for instantiation describes a capture-avoiding substitution operation that replaces all free variables simultaneously. Thus, we may also refer to these substitutions as parallel substitutions. It is based on the ideas introduced in [26] and is very close to the actual mechanisation done using the *Autosubst* library.

Using the previous definition of substitution, we could now define the β-reduction rule as

$$(\lambda.M)N \to M[N \cdot id].$$

Another variation we may encounter when formalising $\boldsymbol{\lambda}$-terms using a nameless syntax is the typing system. A similar aproach to our modification of the typing system can be found in [3, Chapter 7]. We formulate the definition of context and derivable sequents in the nameless setting as follows.

Definition 19 (Nameless context)**.** *A nameless context $\Gamma, \Delta, \ldots$ is a finite (possibly empty) sequence of simple types.*

Notation.

$|\Gamma|$ *is used to denote the length of a context Γ;*

Γ_i *is used to denote the ith element of a context Γ, given $i < |\Gamma|$.*

Definition 20 (Typing rules for nameless $\boldsymbol{\lambda}$-terms)**.**

$$\frac{\Gamma_i = A}{\Gamma \vdash i : A}\ \textit{Var} \qquad \frac{A, \Gamma \vdash M : B}{\Gamma \vdash \lambda.M : A \supset B}\ \textit{Abs} \qquad \frac{\Gamma \vdash M : A \supset B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}\ \textit{App}$$

Claim 2. *Structural rules of weakening, contraction and exchange are admissible in this setting.*

We look at the particular case of the weakening rule that corresponds to the incrementation of every index of the nameless $\boldsymbol{\lambda}$-term.

$$\frac{\Gamma \vdash M : A}{B, \Gamma \vdash M[\uparrow] : A}\ \text{Weakening}$$

2.3 Mechanising meta-theory in Rocq

In this section we discuss basic questions arising in the formalisation of syntax with binders, and introduce a *Rocq* library that helps with such task. Additionally, we illustrate how to formalise basic concepts of the simply typed lambda calculus. This will help to understand our main decisions on the mechanisation of meta-theory developed in this dissertation. The multiary versions of the λ-calculus that we are going to introduce will follow closely the basic approach described here with the corresponding adaptations.

2.3.1 The Rocq Prover

The *Rocq Prover* (former *Coq Proof Assistant*) [20] is an interactive theorem prover based on the expressive formal language called the Polymorphic, Cumulative Calculus of Inductive Constructions. This is a tool that helps in the formalisation of mathematical results and that can interact with a human to generate machine-verified proofs. *Rocq* encode propositions as types and proofs for these propositions as programs in λ-calculus, in line with the Curry-Howard isomorphism.

It is arguably a great tool for mechanising meta-theory as it was widely used in the *POPLmark* challenge [5]. Also, this proof assistant provides many libraries to deal with the issue of variable binding, like *Autosubst*, as we will see in the next sections.

We illustrate two examples of simple inductive definitions in *Rocq*: the natural numbers and polymorphic lists.

a) Natural numbers

The natural numbers can be inductively defined as either zero or a successor of a natural number.

```
Inductive nat : Type :=
| O
| S (n: nat).
```

For example, the number 0 is represented by the constructor `O` and number 2 is represented as `S (S O)`. Of course this serves as an internal representation and we will not refer to natural numbers using these constructors. We can also check the induction principle that *Rocq* generates for the natural numbers.

```
nat_ind
  : ∀P : nat → Prop,
    P 0 → (∀n : nat, P n → P (S n)) → ∀n : nat, P n
```

Therefore, if we want to prove that the sum of natural numbers is associative, we can do it using this induction principle as follows.

```
Theorem sum_associativity :
  ∀a b c, a+(b+c) = (a+b)+c.
  Proof.
    intros.
    induction a.
    - (* 0+(b+c) = 0+b+c *)
      simpl.        (* simplify equation *)
      reflexivity. (* now both sides are equal *)
    - (* (a+1)+(b+c) = (a+1)+b+c *)
      simpl.        (* simplify equation *)
      rewrite IHa. (* rewrite with induciton hypothesis *)
      reflexivity. (* now both sides are equal *)
Qed.
```

b) Polymorphic lists

Polymorphic lists are lists whose items have no predefined type. The inductive definition for these lists is available in the *Rocq* standard library (`Library Stdlib.Lists.List`) along with many operations and properties. Their definition is as follows:

```
Inductive list (A: Type) : Type :=
| nil
| cons (u: A) (l: list A).
```

For example, if we wanted to have a type for lists of natural numbers, we could just invoke the type `list nat`. The list `[0,2,1]` is then represented as `cons 0 (cons 2 (cons 1 nil))`.

Here is an useful lemma on lists provided by the *Rocq* library:

```
Lemma map_app f : ∀l l', map f (l++l') = (map f l)++(map f l').
```

This lemma relates two operations on lists:

1. app (abreviated as ++): appends two lists (ex: `[1,2,3]++[4,5] = [1,2,3,4,5]`);
2. `map`: applies a function to every element on the list (ex: `map f [x,y] = [f x, f y]`).

Given their widespread utility, these operations will be often used in parts of our mechanisation.

2.3.2 Syntax with binders

A direct formalisation of the grammar of λ-terms in *Rocq* results in an inductive definition like:

```
Inductive term : Type :=
| Var (x: var)
| Lam (x: var) (t: term)
| App (s: term) (t: term).
```

The question that this and any similar definition raises is: how do we define the `var` type? Following the usual pen-and-paper approach, this type would be a subset of a ”string type”, where a variable is just a placeholder for a name.

Of course this is fine when dealing with proofs and definitions in a paper. To simplify this, we can even take advantage of conventions, like the one referenced above (by Barendregt). However, this approach to define the `var` type becomes rather exhausting when it comes to rigorously define the required syntactical ingredients, including substitution operations.

There are several alternative approaches described in the literature of mechanisation of meta-theory. The *POPLmark* challenge [5] points to the topic of binding as central for discussing the potential of modern-day proof assistants. From the many alternatives, we chose to follow the nameless syntax proposed by de Bruijn. This is because this approach seemed widely used in the mechanisation of meta-theory.

2.3.3 Autosubst library

The *Autosubst* library [26, 25] for the *Rocq Prover* facilitates the formalisation of syntax with binders. It provides the *Rocq Prover* with two kinds of tactics:

1. `derive` tactics that automatically define substitution (and boilerplate definitions for substitution) over an inductively defined syntax;

2. `asimpl` and `autosubst` tactics that provide simplification and direct automation for proofs dealing with substitution lemmas.

The library makes use of some ideas we have already covered up: de Bruijn syntax and parallel substitutions. There is also a more subtle third ingredient: the theory of explicit substitution [1]. This theory is particularly relevant to the implementation of the `asimpl` and `autosubst` tactics and we will not digress much on it. Essentially, our calculus with parallel substitutions forms a model of the σ-calculus and we may simplify our terms with substitutions using the convergent rewriting equations described by this theory.

Taking the naive example of an inductive definition of the λ-terms in *Rocq*, we now display a definition using *Autosubst*.

```
Inductive term: Type :=
| Var (x: var)
| Lam (t: {bind term})
| App (s: term) (t: term) .
```

In the above definition, there are two different annotations: the `var` and `{bind term}` types. We write these annotations to mark our constructors with variables and binders, respectively, in the syntax we want to mechanise. They play an important role in the internal development of the automated `derive` tactics.

We invoke the *Autosubst* classes, automatically deriving the desired instances as follows.

```
Instance Ids_term : Ids term. derive. Defined.
Instance Rename_term : Rename term. derive. Defined.
Instance Subst_term : Subst term. derive. Defined.
Instance SubstLemmas_term : SubstLemmas term. derive. Defined.
```

The first three lines derive the operations necessary to define the (parallel) substitution over a term.

1. Defining the `ids` function that maps every index into the corresponding variable term ($i \mapsto$ `(Var i)`).

2. Defining the `rename` function that instantiates a term under a variable renaming.

3. Defining the `subst` function that instantiates a term under a parallel substitution over (making use of the already `rename` and `ids`).

Finally, there is also the proof for the substitution lemmas. Here, we see the power of this library, as the proofs for these lemmas (for fairly simple syntaxes) can be generated automatically through the `derive` tactic.

2.3.4 Mechanising the simply typed λ-calculus

For this dissertation, we provide our own mechanisation of the simply typed $\boldsymbol{\lambda}$-calculus, as we will need it in chapter 5. The mechanisation is very straightforward and follows closely the examples given in [25, 26].

a) `SimpleTypes.v`

This module only contains the definition for simple types using a unique base type for simplicity. This definition is isolated because it will be used by multiple modules.

```
Inductive type: Type :=
| Base
| Arr (A B: type): type.
```

b) Lambda.v

This module contains the definitions we need for the formalisations dealing with the simply typed $\boldsymbol{\lambda}$-calculcus. The syntax for terms and *Autosubst* definitions were already presented and explained in the prior subsection.

The module then includes the definition for the one step β-reduction (recall Definition 8). This inductive definition mechanises the β-reduction altogether with the compatibility closure steps ($\rightarrow_\beta$).

```
Inductive step : relation term :=
| Step_Beta s s' u : s' = s.[u::ids] →
step (App (Lam s) t) s'
| Step_Abs s s' : step s s' →
step (Lam s) (Lam s')
| Step_App1 s s' t: step s s' →
step (App s t) (App s' t)
| Step_App2 s t t': step t t' →
step (App s t) (App s t').
```

In this definition we already give use to the substitution operation defined using *Autosubst* (found in the `Step_Beta` constructor). The syntax `s.[u::ids]` is just notation for the defined instantiation of term s under a parallel substitution `u::ids`. This substitution corresponds to the example of substitution shown in the previous section $(u \cdot id)$.

The type for `step` is `relation term` (an alias for `term→term→Prop`), as we are using the `Relations` library found in the *Rocq* standard library containing definitions and lemmas for binary relations.

We also have a definition for the mutually inductive predicate mechanising β-normal forms (recall Definition 10).

```
Inductive normal: term → Prop :=
| nLam s : normal s → normal (Lam s)
| nApps s : apps s → normal s
with apps: term → Prop :=
| nVar x : apps (Var x)
| nApp s t : apps s → normal t → apps (App s t).
```

As before, we do not define directly a set NF of λ-terms, but rather an inductive predicate that λ-terms $t \in$ NF satisfy. This will be our standard aproach when mechanising subsets, as the subset itself is the extension of the defined predicate.

However, we have to be careful using mutually inductive predicates (we refer to [8, Chapter 14.1] for a detailed overview on mutually inductive types and their induction principles). If we want to prove certain propositions that proceed by indcution on the structure of a normal term, we need to have a simultaneous induction principle and prove two propositions simultaneously.

```
Scheme sim_normal_ind := Induction for normal Sort Prop
  with sim_apps_ind := Induction for apps Sort Prop.
Combined Scheme mut_normal_ind from sim_normal_ind, sim_apps_ind.
```

We can generate two new induction principles using the `Scheme` command. Then, we can combine both induction principles using the `Combined Scheme` command. We will often use the combined induction principles in our proofs, as mutually inductive types will appear often.

Here follows an example of the proof for Claim 1 using the combined induction principle. We will prove not only the desired claim but also a proposition over the set of normal applications, NA.

```
Theorem nfs_are_irreducible :
  (∀s, normal s → ~exists t, step s t)
  ∧
  (∀s, apps s → ~exists t, step s t).
Proof.
  apply mut_normal_ind ; intros.
  (* applying the combined induction principle *)
  - intro.
    apply H.
    destruct H0 as [t Ht].
    inversion Ht.
    now exists s'.
  - intro.
    apply H.
    destruct H0 as [t Ht].
    now exists t.
  - intro.
    now destruct H.
  - intro.
    destruct H1 as [t0 Ht0].
```

```
    inversion Ht0 ; subst.
    + inversion a.
    + apply H. now exists s'.
    + apply H0. now exists t'.
Qed.
```

The proof uses a couple of tactics that we will not cover in detail. It serves more of an example of how we easily prove a result using the mechanised concepts of one step β-reduction and normal forms.

The last thing our module contains is the typing rules for the $\boldsymbol{\lambda}$-terms (recall Definition 15 and Definition 20).

```
Inductive sequent (Γ: var→type) : term → type → Prop :=
| Ax (x: var) (A: type) :
  Γ x = A → sequent Γ (Var x) A
| Intro (t: term) (A B: type) :
  sequent (A.:Γ) t B → sequent Γ (Lam t) (Arr A B)
| Elim (s t: term) (A B: type) :
  sequent Γ s (Arr A B) → sequent Γ t A → sequent Γ (App s t) B.
```

We directly mechanise the derivability of a sequents using an inductively defined predicate (instead of defining sequents *a priori*).

Furthermore, following the approach in [25], we use infinite contexts (contexts as infinite sequences). That way we can mechanise contexts as functions `var→type` (the type of a parallel substitution object over `type`) and take more advantage of the *Autosubst* definitions and tactics. Of course, in any typing derivation, only a finite part of the (infinite) context is used.

A small illustration of the versatility of this option is in the `Intro` rule, where one can find the context `(A.:Γ)`. This is the same function we encountered when defining the substitution operation for the β-contractum `s.[u.:ids]`.

As claimed (Claim 2) upon the definition of the typing rules for the nameless terms, we can show admissibility for the structural rules of weakening, contraction and exchange. We do this by proving the preservation of renamings (also an idea from [25]), as the mentioned structural rules can be seen as a particular case of index renaming (as we have illustrated with the weakening case).

```
Lemma type_renaming : ∀Γ t A, sequent Γ t A →
  ∀Δ ξ, Γ = (ξ >>> Δ) → sequent Δ t.[ren ξ] A
```

Chapter 3

Multiary λ-calculus and its canonical subsystem

This chapter introduces the main system that was studied in this dissertation: the multiary $\boldsymbol{\lambda}$-calculus ($\boldsymbol{\lambda m}$). We introduce this system as the system $\boldsymbol{\lambda \mathcal{P} h}$ studied in [13, Chapter 3]. This system can also be found as $\boldsymbol{\lambda^m}$ in [14, Section 3], as a subsystem of $\boldsymbol{\lambda J^m}$.

We provide an alternative description for a subsystem of h-normal forms of $\boldsymbol{\lambda m}$ (corresponding to the system $\boldsymbol{\lambda \mathcal{P}}$ found in [13, Chapter 3]). At the end of this chapter one can find a detailed overview of the mechanisation done in this dissertation of the multiary $\boldsymbol{\lambda}$-calculus and subsystems.

3.1 The system λm

First, we introduce some standard definitions for our system, like the grammar for $\boldsymbol{\lambda m}$-terms, a typical append operation on lists and substitution operation.

Definition 21 ($\boldsymbol{\lambda m}$-expressions)**.** *The $\boldsymbol{\lambda m}$-terms are simultaneously defined with $\boldsymbol{\lambda m}$-lists by the following grammar:*

$$\begin{array}{rrl} (\boldsymbol{\lambda m}\text{-terms}) & t, u, v ::= & x \mid \lambda x.t \mid t(u, l) \\ (\boldsymbol{\lambda m}\text{-lists}) & l ::= & [] \mid u :: l. \end{array}$$

We will refer to the union of $\boldsymbol{\lambda m}$-terms and $\boldsymbol{\lambda m}$-lists as $\boldsymbol{\lambda m}$-expressions.

Definition 22 (Append)**.** *The append of two $\boldsymbol{\lambda m}$-lists, $l + l'$, is defined recursively on l as follows:*

$$[] + l' = l',$$
$$(u :: l) + l' = u :: (l + l').$$

Definition 23 (Substitution for $\boldsymbol{\lambda m}$-expressions)**.** *The substitution of a variable x by a $\boldsymbol{\lambda m}$-term v is mutually defined by recursion over $\boldsymbol{\lambda m}$-expressions as follows:*

$$\begin{aligned} & x[x := v] = v; \\ & y[x := v] = y, \text{ with } x \neq y; \\ & (\lambda y.t)[x := v] = \lambda y.(t[x := v]); \\ & t(u, l)[x := v] = t[x := v](u[x := v], l[x := v]); \\ & [][x := v] = []; \\ & (u :: l)[x := v] = u[x := v] :: l[x := v]. \end{aligned}$$

Definition 24 (Reduction rules for $\boldsymbol{\lambda m}$-terms)**.** *Consider the following reduction rules for* $\boldsymbol{\lambda m}$*-terms.*

$$\begin{array}{rl} (\beta_1) & (\lambda x.t)(u, []) \to t[x := u] \\ (\beta_2) & (\lambda x.t)(u, v :: l) \to t[x := u](v, l) \\ (h) & t(u,l)(u', l') \to t(u, l + (u' :: l')) \end{array}$$

Of course, one may also interpret the given rules as binary relations on $\boldsymbol{\lambda m}$*-terms. That way, we can define a relation* β *as the relation* $\beta_1 \cup \beta_2$ *and analogously a relation* βh *as the relation* $\beta \cup h$*.*

Definition 25 (Compatible Relation)**.** *Let* R *and* R' *be two binary relations on* $\boldsymbol{\lambda m}$*-terms and* $\boldsymbol{\lambda m}$*-lists respectively. We say they are compatible when they satisfy:*

$$\frac{(t,t') \in R}{(\lambda x.t, \lambda x.t') \in R} \qquad \frac{(t,t') \in R}{(t(u,l), t'(u,l)) \in R} \qquad \frac{(u,u') \in R}{(t(u,l), t(u',l)) \in R} \qquad \frac{(l,l') \in R'}{(t(u,l), t(u,l')) \in R}$$

$$\frac{(u,u') \in R}{(u :: l, u' :: l) \in R'} \qquad \frac{(l,l') \in R'}{(u :: l, u :: l') \in R'}$$

Notation. *We will use the same notation for relations introduced in chapter 2. As the compatible closure induces two relations, one on terms and the other on lists, we will use the already familiar notation* $\to_R$ *for both these relations as we can get out of the context which one is being referenced.*

Then, we will have the induced relations $\to_\beta$ and $\to_{\beta h}$ on $\boldsymbol{\lambda m}$-expressions. We may also refer to the multistep analogous relations $\twoheadrightarrow_\beta$ and $\twoheadrightarrow_{\beta h}$.

We turn now our attention to the typing system of $\boldsymbol{\lambda m}$. Given that $\boldsymbol{\lambda m}$ has two syntactic categories of expressions, its typing system will deal with two different kinds of sequents.

Definition 26 (Sequent)**.** *A sequent on terms* $\Gamma \vdash t : A$ *is a triple of a context, a* $\boldsymbol{\lambda m}$*-term and a simple type. A sequent on lists* $\Gamma; A \vdash l : B$ *is a quadruple of a context, a simple type, a* $\boldsymbol{\lambda m}$*-list and another simple type.*

Definition 27 (Typing Rules for $\boldsymbol{\lambda m}$-terms)**.**

$$\frac{}{x : A, \Gamma \vdash x : A}\ \textit{Var} \qquad \frac{x : A, \Gamma \vdash t : B}{\Gamma \vdash \lambda x.t : A \supset B}\ \textit{Abs}$$

$$\frac{\Gamma \vdash t : A \supset B \qquad \Gamma \vdash u : A \qquad \Gamma; B \vdash l : C}{\Gamma \vdash t(u,l) : C}\ \textit{mApp}$$

$$\frac{}{\Gamma; A \vdash [] : A}\ \textit{Nil} \qquad \frac{\Gamma \vdash u : A \qquad \Gamma; B \vdash l : C}{\Gamma; A \supset B \vdash u :: l : C}\ \textit{Cons}$$

As usual a sequent derivation is a tree-like structure, with root being the derived sequent and leaves being instances of the axioms (Var-rule or Nil-rule).

Now follow two necessary lemmas for the result of subject reduction that state the admissibility of substitution and append operations.

Lemma 1 (Substitution is admissible)**.** *The following rules are admissible:*

$$\frac{\Gamma, x : B \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash t[x := u] : A} \qquad\qquad \frac{\Gamma, x : B\,; C \vdash l : A \qquad \Gamma \vdash u : B}{\Gamma; C \vdash l[x := u] : A}.$$

Proof. The proof proceeds by simultaneous induction on the structure of the term t and list l. ☐

Lemma 2 (Append is admissible)**.** *The following rule is admissible:*

$$\frac{\Gamma; C \vdash l : B \qquad \Gamma; B \vdash l' : A}{\Gamma; C \vdash l + l' : A}.$$

Proof. The proof proceeds by induction on the structure of l. ☐

Subject reduction then states that any given term preserves its type upon βh reduction.

Theorem 1 (Subject reduction)**.** *Given $\boldsymbol{\lambda m}$-terms t and t', the follwing holds:*

$$\Gamma \vdash t : A \;\wedge\; t \to_{\beta h} t' \implies \Gamma \vdash t' : A.$$

Proof. The proof proceeds by induction on the structure of the relation $\to_{\beta h}$.

Lemma 1 is used to prove the case where $(t, t') \in \beta$.

Lemma 2 is used to prove the case $(t, t') \in h$. ☐

Corollary 1 (Multistep subject reduction)**.** *Given $\boldsymbol{\lambda m}$-terms t and t', the follwing holds:*

$$\Gamma \vdash t : A \;\wedge\; t \twoheadrightarrow_{\beta h} t' \implies \Gamma \vdash t' : A.$$

Proof. Trivial. ☐

Other classical results from $\boldsymbol{\lambda}$-calculus like confluency and strong normalisation could also be proved for system $\boldsymbol{\lambda m}$, but are not covered in this dissertation.

3.2 The canonical subsystem

The canonical subsystem is a system within $\boldsymbol{\lambda m}$ containing only terms in h-normal form. In this section we see how to equip canonical terms with an appropriate notion of β-reduction and appropriate typing rules.

Definition 28 (h-normal form)**.** *We say that a* $\boldsymbol{\lambda m}$*-term* t *is in* h*-normal form when there exists no* $\boldsymbol{\lambda m}$*-term* t' *such that*

$$t \to_h t'.$$

Definition 29 (Canonical expressions)**.** *We inductively define the subsets of* $\boldsymbol{\lambda m}$*-terms and* $\boldsymbol{\lambda m}$*-lists, respectively* Can *and* $CanList$*, as follows:*

$$\frac{}{x \in Can} \qquad \frac{t \in Can}{\lambda x.t \in Can} \qquad \frac{u \in Can \qquad l \in CanList}{x(u,l) \in Can} \qquad \frac{t \in Can \qquad u \in Can \qquad l \in CanList}{(\lambda x.t)(u,l) \in Can}$$

$$\frac{}{[] \in CanList} \qquad \frac{u \in Can \qquad l \in CanList}{u :: l \in CanList}$$

$\boldsymbol{\lambda m}$-terms $t \in Can$ are also called canonical terms. Analogously, $\boldsymbol{\lambda m}$-lists $l \in CanList$ are called canonical lists. Canonical expressions will refer to the set $Can \cup CanList$.

Similar to what was done in chapter 2, we leave a claim stating that the canonical terms are exactly the $\boldsymbol{\lambda m}$-terms in h-normal form.

Claim 3. *Given a* $\boldsymbol{\lambda m}$*-term* t*, the following are equivalent:*

(i) $t \in Can.$

(ii) t *is in* h*-normal form.*

Now, we will describe how the canonical terms generate a subsystem.

First, we define the function $app : Can \times Can \times CanList \to Can$ that will behave as a multiary application constructor closed for the canonical terms.

Definition 30. *Given* $t, u \in Can$ *and* $l \in CanList$*, the operation* $app(t, u, l)$ *is defined by the following equations:*

$$\begin{aligned}
app(x, u, l) &= x(u, l),\\
app(\lambda x.t, u, l) &= (\lambda x.t)(u, l),\\
app(x(u', l'), u, l) &= x(u', l' + (u :: l))\\
app((\lambda x.t)(u', l'), u, l) &= (\lambda x.t)(u', l' + (u :: l)).
\end{aligned}$$

Lemma 3. *Given* $t, u \in Can$ *and* $l \in CanList$*,*

$$t(u, l) \twoheadrightarrow_h app(t, u, l) \qquad \textit{(in } \boldsymbol{\lambda m}\textit{)}.$$

Proof. The proof proceeds easily by inspection of term t.

For the cases where t is not an application, we have an equality. □

Then, we can define a function that collapses $\boldsymbol{\lambda m}$-terms to their h-normal form.

Definition 31. *Consider the following map h : $\boldsymbol{\lambda m}$-terms $\to Can$, recursively defined as follows:*

$$
\begin{aligned}
h(x) &= x \\
h(\lambda x.t) &= \lambda x.h(t) \\
h(t(u,l)) &= app(h(t),h(u),h(l)) \\
h([]) &= [] \\
h(u :: l) &= h(u) :: h(l).
\end{aligned}
$$

Proposition 1 (Map h performs $\twoheadrightarrow_h$)**.** *For every $\boldsymbol{\lambda m}$-term t,*

$$t \twoheadrightarrow_h h(t),$$

and also, for every $\boldsymbol{\lambda m}$-list l,

$$l \twoheadrightarrow_h h(l).$$

Proof. The proof proceeds easily by simultaneous induction on the structure of $\boldsymbol{\lambda m}$-expressions. As map h is defined using app, Lemma 3 is crucial for the case where t is an application. □

With the following auxiliary result we can easily prove the confluence of $\twoheadrightarrow_h$.

Lemma 4 (Map h collapses $\to_h$)**.** *For every $\boldsymbol{\lambda m}$-terms t, t',*

$$t \to_h t' \implies h(t) = h(t').$$

Proof. The proof proceeds easily by induction on the structure of $t \to_h t'$. □

Corollary 2 (Confluence of $\twoheadrightarrow_h$)**.** *For every $\boldsymbol{\lambda m}$-terms t, t_1, t_2,*

$$
\begin{tikzcd}
& t \arrow[dl, two heads, "h"'] \arrow[dr, two heads, "h"] \arrow[dd, dashed, two heads, "h"] & \\
t_1 \arrow[dr, two heads, "h"'] & & t_2 \arrow[dl, two heads, "h"] \\
& h(t) &
\end{tikzcd}
$$

Proof. Immediate by Lemma 4 and Proposition 1. □

The following theorem states that the canonical terms are invariant or fixpoints for map h. Another way to look at this result is by saying that h is surjective.

Proposition 2 (Invariance of canonical terms by h)**.** *For every $t \in Can$,*

$$h(t) = t,$$

and also, for every $l \in CanList$,

$$h(l) = l.$$

Proof. The proof proceeds easily by simultaneous induction on the structure of canonical expressions. $\square$

For the purpose of defining a subsystem of $\boldsymbol{\lambda m}$, we will see how to induce a reduction relation for these canonical expressions given a reduction relation on $\boldsymbol{\lambda m}$-expressions.

Definition 32 (Canonical relation closure)**.** *Let R and R' be two binary relations on $\boldsymbol{\lambda m}$-terms and $\boldsymbol{\lambda m}$-lists respectively. We inductively define the relations R_c and R'_c, on canonical terms and lists respectively, as follows:*

$$\frac{(t, t') \in R}{(h(t), h(t')) \in R_c} \qquad \frac{(l, l') \in R'}{(h'(l), h'(l')) \in R'_c}.$$

We call canonical relation closure of R and R' to the induced relations R_c and R'_c.

This definition allows us to define a concept of β-reduction for the canonical terms, namely $(\rightarrow_\beta)_c$, derived from the relation $\rightarrow_\beta$ in $\boldsymbol{\lambda m}$. But this definition tells us little about the relation itself ...an interesting question is: how does a β-reduction (as in the previous definition) behave on the canonical terms?

Given $t, u \in Can$, lets see how to reduce $(\lambda x.t)(u, [])$. The definition of $(\rightarrow_\beta)_c$ stipulates:

$$\frac{(\lambda x.t)(u, []) \rightarrow_\beta t[x := u]}{h((\lambda x.t)(u, [])) \ (\rightarrow_\beta)_c \ h(t[x := u])}$$

Given that $t, u \in Can$, we get that $(\lambda x.t)(u, []) \in Can$. Therefore, from Proposition 2, we get $(\lambda x.t)(u, []) \ (\rightarrow_\beta)_c \ h(t[x := u])$.

Furthermore, from this definition, we could even prove certain properties of $(\rightarrow_\beta)_c$, such as:

$$\frac{t \ (\rightarrow_\beta)_c \ t'}{\lambda x.t \ (\rightarrow_\beta)_c \ \lambda x.t'}$$

This follows from "inverting" $t \ (\rightarrow_\beta)_c \ t'$. Firstly, one observes that there exist $\boldsymbol{\lambda m}$-terms u, u' such that $h(u) = t$ and $h(u') = t'$ and $u \rightarrow_\beta u'$. Then,

$$\cfrac{\cfrac{u \rightarrow_\beta u'}{\lambda x.u \rightarrow_\beta \lambda x.u'} \ \text{(compatibility of } \rightarrow_\beta\text{)}}{h(\lambda x.u) \ (\rightarrow_\beta)_c \ h(\lambda x.u')} \ \text{(Definition 32)}$$

Lastly, simplifying h and rewriting $h(u)$ and $h(u')$, we conclude that $\lambda x.t \ (\rightarrow_\beta)_c \ \lambda x.t'$.

We now conclude the presentation of the canonical subsystem in $\boldsymbol{\lambda m}$, by equiping canonical expressions with a typing relation, in the same spirit of Definition 32.

Definition 33 (Canonical typing closure)**.** *We define the derivable sequents for canonical expressions as follows:*

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash_c h(t) : A} \qquad\qquad \frac{\Gamma; A \vdash l : B}{\Gamma; A \vdash_c h'(l) : B}$$

Also, from the previous definition, we may ask similar questions to those asked above about β-reduction for canonical expressions. For example, given $t \in Can$, is the following rule admissible?

$$\frac{x : A, \Gamma \vdash_c t : B}{\Gamma \vdash_c \lambda x.t : A \supset B}$$

By inverting our assumption of $x : A, \Gamma \vdash_c t : B$, we get that there exists t', such that $h(t') = t$ and $x : A, \Gamma \vdash t' : B$ is derivable in $\boldsymbol{\lambda m}$. Then,

$$\cfrac{\cfrac{x : A, \Gamma \vdash t' : B}{\Gamma \vdash \lambda x.t' : A \supset B}\ \text{Lam}}{\Gamma \vdash_c h(\lambda x.t') : A \supset B}\ \text{(Definition 33)}$$

And again, simplifying and rewriting h, we have derived the sequent $\Gamma \vdash_c \lambda x.t : A \supset B$.

Our presentation of the canonical subsystem of $\boldsymbol{\lambda m}$ does not exaclty coincide with system $\boldsymbol{\lambda \mathcal{P}}$ from [13, Chapter 3.1]. We define a subsystem of $\boldsymbol{\lambda m}$ by restricting our syntax of expressions via a map h. Then, we introduce reduction and a typing relation by exclusively using map h. Trivially, we get a subsystem with appropriate notions of reduction and typing but still preserving their expected behaviour (as seen above).

In our work, motivated by the task of mechanisation, we distinguish between a subsystem of $\boldsymbol{\lambda m}$ in the sense we have described before and an isomorphic system with its own syntax, substitution, reduction and typing rules (this is the system $\vec{\boldsymbol{\lambda}}$ that will be covered in chapter 4). We explain some details and motivations for this at the end of the next section.

3.3 Mechanisation in Rocq

The mechanisation of the system $\boldsymbol{\lambda m}$ also crucially relies on the *Autosubst* library, and essentially follows the style adopted for the mechanisation of the simply typed $\boldsymbol{\lambda}$-calculus we have seen in chapter 2.

3.3.1 `LambdaM.v`

This module contains the necessary definitions for the various aspects of the formalisation of system $\boldsymbol{\lambda m}$ performed in this dissertation. The inductive type for the syntax of $\boldsymbol{\lambda m}$-terms is as follows.

```
Inductive term: Type :=
| Var (x: var)
| Lam (t: {bind term})
| mApp (t: term) (u: term) (l: list term).
```

Note that the definition for λm-lists is hidden under the polymorphic list type `list term`. We explain the reason for more details for this option at the end of this section.

To mechanise the reduction relations, we first defined the notion of compatibility for binary relations on λm-expressions (as in Definition 25) and then define the base step relations β_1, β_2 and h separately. That way we can distinguish between the notions of compatible closure of a base relation and of a relation being compatible. This aproach is more elaborated than the one presented for the simply typed λ-calculus and we also get into more details about these decisions at the end of this section.

```
Inductive β1: relation term :=
| Step_Beta1 (t: {bind term}) (t' u: term) :
  t' = t.[u .: ids] → β1 (mApp (Lam t) u []) t'.

Inductive β2: relation term :=
| Step_Beta2 (t: {bind term}) (t' u v: term) l :
  t' = t.[u .: ids] → β2 (mApp (Lam t) u (v::l)) (mApp t' v l).

Inductive H: relation term :=
| Step_H (t u u': term) l l' l'' :
  l'' = l ++ (u'::l') → H (mApp (mApp t u l) u' l') (mApp t u l'').

Definition step := comp (union _ (union _ β1 β2) H).
Definition step' := comp' (union _ (union _ β1 β2) H).

Definition multistep := clos_refl_trans_1n _ step.
Definition multistep' := clos_refl_trans_1n _ step'.
```

Here, `comp` and `comp'` are the polymorphic relations on λm-expressions that induce the compatibility closure. We also note the use of the `clos_refl_trans_1n` polymorphic relation provided by the *Rocq Prover* libraries that induces the reflexive and transitive closure of a given binary relation.

In this module, we also find the formalisation of the typing relation for λm, through an inductively defined relation, much in the style of what was done for the simply typed λ-calculus.

```
Inductive sequent (Γ: var→type) : term → type → Prop :=
```

```
| varAxiom (x: var) (A: type) :
  Γ x = A → sequent Γ (Var x) A
| Right (t: term) (A B: type) :
  sequent (A .: Γ) t B → sequent Γ (Lam t) (Arr A B)
| HeadCut (t u: term) (l: list term) (A B C: type) :
  sequent Γ t (Arr A B) → sequent Γ u A → list_sequent Γ B l C →
  sequent Γ (mApp t u l) C
with list_sequent (Γ:var→type) : type → (list term) → type → Prop :=
| nilAxiom (C: type) : list_sequent Γ C [] C
| Lft (u: term) (l: list term) (A B C:type) :
  sequent Γ u A → list_sequent Γ B l C →
  list_sequent Γ (Arr A B) (u :: l) C.
```

3.3.2 `TypePreservation.v`

This module contains the proof of the subject reduction theorem (Theorem 1) and necessary lemmas to prove it (recall Lemma 1).

```
Theorem type_preservation :
  (∀t t', step t t' → ∀Γ A, sequent Γ t A → sequent Γ t' A)
  ∧
  (∀l l', step' l l' → ∀Γ A B, list_sequent Γ A l B →
                                list_sequent Γ A l' B).
```

Using *Autosubst*, we have to prove not only the preservation of types by the substitution operation but also by renamings. We prove these results using the techniques in the tutorial [25].

```
Lemma type_renaming :
  ∀Γ,
    (∀t A, sequent Γ t A →
       ∀Δ ξ, Γ = (ξ >>> Δ) → sequent Δ t.[ren ξ] A)
    ∧
    (∀A l B, list_sequent Γ A l B →
       ∀Δ ξ, Γ = (ξ >>> Δ) → list_sequent Δ A l..[ren ξ] B).
...
Lemma type_substitution :
  ∀Γ,
    (∀t A, sequent Γ t A →
```

```
    ∀σ Δ, (∀x, sequent Δ (σ x) (Γ x)) → sequent Δ t.[σ] A)
  ∧
  (∀A l B, list_sequent Γ A l B →
    ∀σ Δ, (∀x, sequent Δ (σ x) (Γ x)) → list_sequent Δ A l..[σ] B).
```

For what is worth, we could prove a simpler statement (similar to Lemma 1) to formalise the subject reduction theorem. Such lemma would look like (without the proposition for lists):

```
Lemma weak_type_substitution Γ t A :
  sequent (B.:Γ) t A → sequent Γ u B →
  sequent Γ t.[u.:σ] A).
```

The used *Autosubst* aproach takes this notion of well-typed substitutions or context morphisms (see [26, Chapter 4]) to generalise these lemmas.

As already mentioned, we use the combined induction principles (for λm-expressions) to prove the statements that are declared using a conjunction on terms and lists.

3.3.3 `IsCanonical.v`

This module contains the necessary definitions for the formalisation of the canonical subsystem of λm.

First, we define a predicate `is_canonical` that constructively defines the canonical expressions in the style of Definition 29.

```
Inductive is_canonical: term → Prop :=
| cVar (x: var) :
  is_canonical (Var x)
| cLam (t: {bind term}) :
  is_canonical t → is_canonical (Lam t)
| cVarApp (x: var) (u: term) (l: list term) :
  is_canonical u → is_canonical_list l →
  is_canonical (mApp (Var x) u l)
| cLamApp (t: {bind term}) (u: term) (l: list term) :
  is_canonical t → is_canonical u → is_canonical_list l →
  is_canonical (mApp (Lam t) u l)
with is_canonical_list: list term → Prop :=
| cNil : is_canonical_list []
| cCons (u: term) (l: list term) :
  is_canonical u → is_canonical_list l →
  is_canonical_list (u::l).
```

The module then contains defintions for the app operation (called `capp` because append of lists in *Rocq* is already called `app`) and map h.

```
Definition capp (v u: term) (l: list term) : term :=
  match v with
  | Var x          ⇒ mApp v u l
  | Lam t          ⇒ mApp v u l
  | mApp t u' l' ⇒ mApp t u' (l' ++ (u::l))
  end.

Fixpoint h (t: term) :=
  match t with
  | Var x        ⇒ Var x
  | Lam t        ⇒ Lam (h t)
  | mApp t u l ⇒ capp (h t) (h u) (map h l)
  end.
```

In our definition, `map h` (which calls the `map` function from the `List` library) behaves exactly as the intended map h when applied to lists.

In the *Rocq Prover*, we need to formally prove that the app operation and map h are closed for canonical terms. Note that in the case of our description of the subsystem in the previous section, it is easy to informally argue about this. For example, in our mechanisation, we have the following lemma.

```
Lemma capp_is_canonical t u l :
  is_canonical t → is_canonical u → is_canonical_list l →
  is_canonical (capp t u l).
```

Then, we prove all the lemmas, propositions and theorems presented in the description of the canonical subsystem. As an example, we show the mechanisation of Proposition 2.

```
Proposition h_fixpoints :
  (∀t, is_canonical t → h t = t)
  ∧
  (∀l, is_canonical_list l → map h l = l).
Proof.
  apply mut_is_canonical_ind ;
    intros ; asimpl ; repeat f_equal ; auto.
Qed.
```

In this proof we use the **auto** tactic to facilitate our work. For routine proofs, we often found success when using these automated tactics.

The module ends with definitions for the reduction relation (recall Definition 32) and typing rules (recall Definition 33) for the canonical subsystem.

```
Inductive canonical_relation
  (R: relation term) : relation term :=
| Step_CanTerm t t' : R t t' → canonical_relation R (h t) (h t').
Inductive canonical_list_relation
  (R: relation (list term)) : relation (list term) :=
| Step_CanList l l' : R l l' → canonical_list_relation R (map h l) (map h l').

Definition step_can := canonical_relation step_beta.
Definition step_can' := canonical_list_relation step_beta'.
...
Inductive canonical_sequent (Γ: var→type) :
  term → type → Prop :=
| Seq_CanTerm t A : sequent Γ t A → canonical_sequent Γ (h t) A.
Inductive canonical_list_sequent (Γ: var→type) :
  type → list term → type → Prop :=
| Seq_CanList l A B : list_sequent Γ A l B →
                      canonical_list_sequent Γ A (map h l) B.
```

3.3.4 A closer look at the mechanisation

In this part we take a closer look at some particular aspects of the mechanisation that deserve more attention. The purpose is to show how some other options could arise and justify aspects of our approach that may look unusual.

a) Mutually inductive types vs nested inductive types

Creating a mutually inductive type for the syntax of λm in *Rocq* would be a simple task:

```
Inductive term: Type :=
| Var (x: var)
| Lam (t: {bind term})
| mApp (t: term) (u: term) (l: list)
```

```
with list: Type :=
| Nil
| Cons (u: term) (l: list).
```

However, as reported in the final section of [26], *Autosubst* offers no support for mutually inductive definitions. The `derive` tactic would not generate the desired instances for the `Rename` and `Subst` classes, failing to iterate through the customized list type.

As we tried to keep the decision of using *Autosubst*, there were two possible directions:

1. manually define every instance required and prove substitution lemmas;

2. remove the mutual dependency in the term definition.

The first formalisation attempts followed the first option. This meant that everything *Autosubst* could provide automatically was done by hand. For this, we closely followed the definitions given in [26].

After some closer inspection of the library source code, we found that there was native support for the use of types depending on polymorphic lists. This way, there was no need of having a mutual inductive type for our terms.

The downside of using nested inductive types in the *Rocq Prover* is the generated induction principles. This issue is already well documented in [8, Chapter 14.3]. With this approach, we need to provide the dedicated induction principles to the proof assistant, presented below.

```
Section dedicated_induction_principle.
  Variable P : term → Prop.
  Variable Q : list term → Prop.
  Hypothesis HVar : ∀x, P (Var x).
  Hypothesis HLam : ∀t: {bind term}, P t → P (Lam t).
  Hypothesis HmApp : ∀t u l, P t → P u → Q l → P (mApp t u l).
  Hypothesis HNil : Q [].
  Hypothesis HCons : ∀u l, P u → Q l → Q (u::l).

  Proposition sim_term_ind : ∀t, P t.
  Proof.
    fix rec 1. destruct t.
    - now apply HVar.
    - apply HLam. now apply rec.
    - apply HmApp.
      + now apply rec.
      + now apply rec.
```

```
      + assert (∀l, Q l). {
            fix rec' 1. destruct l0.
            - apply HNil.
            - apply HCons.
              + now apply rec.
              + now apply rec'. }
        now apply H.
  Qed.

  Proposition sim_list_ind : ∀l, Q l.
  Proof.
    fix rec 1. destruct l.
    - now apply HNil.
    - apply HCons.
      + now apply sim_term_ind.
      + now apply rec.
  Qed.
End dedicated_induction_principle.
```

b) Formalising a compatible closure

Defining reduction relations in λ-calculi like systems always involve the notion of compatiblity closure, as we want to allow reduction to happen at the level of subterms.

We took inspiration from the definitions in the `Relations` libraries of the *Rocq Prover*. This library provides many definitions on binary relations. For example, there is a predicate that transitive relations satisfy (in `Relation_Definitions`) and there is also a higher order relation that constructs the transitive closure of a given relation (in `Relation_Operations`).

```
Definition transitive : Prop := ∀x y z:A, R x y → R y z → R x z.
...
Inductive clos_trans (x: A) : A → Prop :=
| t_step (y:A) : R x y → clos_trans x y
| t_trans (y z:A) : clos_trans x y → clos_trans y z → clos_trans x z.
```

We followed these definitions to define compatibility notions for the system λm in a modular way. We define the compatible closure from a given base relation on λm-terms as follows:

```
Section Compatibilty.
```

```
  Variable base : relation term.

  Inductive comp : relation term :=
  | Comp_Lam (t t': {bind term}) : comp t t' →
                                  comp (Lam t) (Lam t')
  | Comp_mApp1 t t' u l : comp t t' →
                          comp (mApp t u l) (mApp t' u l)
  | Comp_mApp2 t u u' l : comp u u' →
                          comp (mApp t u l) (mApp t u' l)
  | Comp_mApp3 t u l l' : comp' l l' →
                          comp (mApp t u l) (mApp t u l')
  | Step_Base t t' : base t t' → comp t t'
  with comp' : relation (list term) :=
  | Comp_Head u u' l : comp u u' → comp' (u::l) (u'::l)
  | Comp_Tail u l l' : comp' l l' → comp' (u::l) (u::l').

  Scheme sim_comp_ind := Induction for comp Sort Prop
    with sim_comp_ind' := Induction for comp' Sort Prop.
  Combined Scheme mut_comp_ind from sim_comp_ind, sim_comp_ind'.
End Compatibilty.
```

Then, we also define a record type that contains the necessary predicates to be satisfied by a compatible relation.

```
Section IsCompatible.
  Variable R : relation term.
  Variable R' : relation (list term).

  Record is_compatible := {
      comp_lam : ∀t t': {bind term}, R t t' → R (Lam t) (Lam t') ;
      comp_mApp1 : ∀t t' u l, R t t' → R (mApp t u l) (mApp t' u l) ;
      comp_mApp2 : ∀t u u' l, R u u' → R (mApp t u l) (mApp t u' l) ;
      comp_mApp3 : ∀t u l l', R' l l' → R (mApp t u l) (mApp t u l') ;
      comp_head : ∀u u' l, R u u' → R' (u :: l) (u' :: l) ;
      comp_tail : ∀u l l', R' l l' → R' (u :: l) (u :: l')
    }.
End IsCompatible.
```

From these modular definitions, we can prove some interesting (yet bureaucratic) results, like:

```
Theorem comp_is_compatible B : is_compatible (comp B) (comp' B).
Proof.
  split ; autounfold ; intros ; constructor ; assumption.
Qed.

Theorem clos_refl_trans_pres_comp :
  ∀R R', is_compatible R R' →
    is_compatible (clos_refl_trans_1n _ R) (clos_refl_trans_1n _ R').
Proof.
  intros R R' H. destruct H.
  split ; intros ; induction H ; econstructor ; eauto.
Qed.
```

This theorem states that if we have a compatible relation, its reflexive and transitive closure is still compatible.

An advantage of these modular definitions is that we can use them to increase automation in our proofs. In the main theorem that we prove in the next chapter (bellow is part of it, named `conservativeness2`), our proof starts by adding every compatibility step to our context. As the auto tactic tries to match hypothesis in the context with the goal, the compatibility steps are then covered automatically.

```
Lemma conservativeness2 :
  (∀(t t': LambdaM.term), LambdaM.step t t' →
    Canonical.multistep (p t) (p t'))
  ∧
  (∀(l l': list LambdaM.term), LambdaM.step' l l' →
    Canonical.multistep' (map p l) (map p l')).
Proof.
  pose Canonical.multistep_is_compatible as H.
  destruct H. (* unpacking record type *)
  apply LambdaM.mut_comp_ind ; intros ; asimpl ; auto.
  ...
```

c) Formalising a subsystem

A relevant part of our work was to find simple representations for subsystems in the proof assistant.

As we pointed out, the formalisation we have done for the canonical subsystem of $\boldsymbol{\lambda m}$ is non standard. These ideas were motivated by the task of mechanising such subsystem.

Formalising the subset of terms using a predicate is an obvious way to proceed. But we would also like to have a dedicated type for the extension of that predicate rather than just the predicate itself. The *Rocq Prover* provides such types, known as subset types (we refer to [8, Chapter 9.1]). Although these subset types are exactly what we wanted, they do not give us a great advantage on mechanisation tasks. Using subset types rapidly becomes exhausting because of the need to always provide proof objects in every definition.

As an example, trying to define the one step β-relation as in [13, Chapter 3.1] for the canonical subsystem mechanised using subset types, we would get (supposing we had a mechanised substitution operation):

```
Definition canonical := { u: term | is_canonical u }.
Definition canonical_list := { l: list term | is_canonical_list l }.
...
Inductive can_step : canonical → canonical → Prop :=
| cStep_Beta1 (t u: term) (it: is_canonical t) (iu: is_canonical u)
  (t': canonical) i:
  i = (cLamApp t u []) it iu cNil →
  t' = (exist _ t it).[(exist _ u iu) .: ids] →
  can_step (exist _ (mApp (Lam t) u []) i) t'
...
| cStep_Lam t t' it it' i1 i2 :
  i1 = (cLam t) it →
  i2 = (cLam t') it' →
  can_step (exist _ t it) (exist _ t' it') →
  can_step (exist _ (Lam t) i1) (exist _ (Lam t') i2)
...
```

So far, our aproach on the formalisation of the canonical subsystem of $\boldsymbol{\lambda m}$ was to view it according to map h, that is, defining reduction and typification using this map. However, we may also define a self-contained version of the canonical subsystem with its own syntax and definitions (in the spirit of [13, Chapter 3.1]). Then, we may prove that both representations are in fact isomorphic. That is the goal for chapter 4.

Chapter 4

Canonical λ-calculus

In this chapter we present a system that we give the name of canonical $\boldsymbol{\lambda}$-calculus ($\vec{\boldsymbol{\lambda}}$). The naming of this system is motivated by two reasons. On the one hand, it is isomorphic to the canonical subsystem of $\boldsymbol{\lambda m}$ seen in the previous chapter. On the other hand, it is also isomorphic to the simply typed $\boldsymbol{\lambda}$-calculus.

The system $\vec{\boldsymbol{\lambda}}$ is a self-contained representation of the canonical subsystem of $\boldsymbol{\lambda m}$ (one can notice the similarities between the definitions for system $\boldsymbol{\lambda m}$). We will give a complete proof for this isomorphism in the second section of this chapter. In the third section we give the proof for the theorem of conservativeness, stating that $\boldsymbol{\lambda m}$ is a conservative extension of $\vec{\boldsymbol{\lambda}}$. The isomorphism between system $\vec{\boldsymbol{\lambda}}$ and the simply typed $\boldsymbol{\lambda}$-calculus is left for chapter 5.

4.1 The system $\vec{\lambda}$

Definition 34 ($\vec{\boldsymbol{\lambda}}$-expressions)**.** *The $\vec{\boldsymbol{\lambda}}$-terms are simultaneously defined with $\vec{\boldsymbol{\lambda}}$-lists by the following grammar:*

$$
\begin{array}{rrl}
(\vec{\boldsymbol{\lambda}}\textit{-terms}) & t,u ::= & var(x) \mid \lambda x.t \mid app_v(x,u,l) \mid app_\lambda(x.t,u,l) \\
(\vec{\boldsymbol{\lambda}}\textit{-lists}) & l ::= & [] \mid u :: l.
\end{array}
$$

We will refer to the union of $\vec{\boldsymbol{\lambda}}$-terms and $\vec{\boldsymbol{\lambda}}$-lists as $\vec{\boldsymbol{\lambda}}$-expressions.

Remark. *The $\vec{\boldsymbol{\lambda}}$-terms have two different binding constructors: $\lambda x.t$ and $app_\lambda(x.t,u,l)$. In both constructors, every occurrence of the variable x in subterms $var(x)$ of the term t is bound (and not free). System $\vec{\boldsymbol{\lambda}}$ has in $app_\lambda(x.t,u,l)$ a dedicated constructor for the multiary application $(\lambda x.t)(u,l)$ of system $\boldsymbol{\lambda m}$.*

Definition 35. *Given $\vec{\boldsymbol{\lambda}}$-terms t,u and a $\vec{\boldsymbol{\lambda}}$-list l, the operation $t@(u,l)$ calculates a $\vec{\boldsymbol{\lambda}}$-term defined by the following equations:*

$$
\begin{aligned}
var(x)@(u,l) &= app_v(x,u,l), \\
(\lambda x.t)@(u,l) &= app_\lambda(x.t,u,l), \\
app_v(x,u',l')@(u,l) &= app_v(x,u',l' + (u :: l)) \\
app_\lambda(x.t,u',l')@(u,l) &= app_\lambda(x.t,u',l' + (u :: l)),
\end{aligned}
$$

where the list append, $l + l'$, has the expected bahaviour (as in $\boldsymbol{\lambda m}$).

Now follows a "strange" definition for the substitution operation, which needs to be careful when dealing with a substitution over a constructor app_v.

Definition 36 (Substitution for $\vec{\lambda}$-expressions)**.** *The substitution of a variable x by a $\vec{\lambda}$-term u is mutually defined by recursion over $\vec{\lambda}$-expressions as follows:*

$$var(x)[x := u] = u;$$
$$var(y)[x := u] = var(y), \text{ with } x \neq y;$$
$$(\lambda y.t)[x := u] = \lambda y.(t[x := u]);$$
$$app_v(x, u', l)[x := u] = u@(u'[x := u], l[x := u]);$$
$$app_v(y, u', l)[x := u] = app_v(y, u'[x := u], l[x := u]), \text{ with } x \neq y;$$
$$app_\lambda(y.t, u', l)[x := u] = app_\lambda(y.t[x := u], u'[x := u], l[x := u]);$$
$$[][x := u] = [];$$
$$(v :: l)[x := u] = v[x := u] :: l[x := u].$$

Definition 37 (Reduction rules for $\vec{\lambda}$-terms)**.**

$$(\beta_1) \qquad app_\lambda(x.t, u, []) \to t[x := u]$$
$$(\beta_2) \qquad app_\lambda(x.t, u, v :: l) \to t[x := u]@(v, l)$$

As before, we look at the previous rules as binary relations on $\vec{\lambda}$-terms and define a relation $\beta = \beta_1 \cup \beta_2$.

Now, we make a small detour dedicated to compatible relations in the context of this system.

Definition 38 (Compatible relation)**.** *Let R and R' be two binary relations on $\vec{\lambda}$-terms and $\vec{\lambda}$-lists respectively. We say they are compatible when they satisfy:*

$$\frac{(t, t') \in R}{(\lambda x.t, \lambda x.t') \in R} \qquad \frac{(t, t') \in R}{(app_\lambda(x.t, u, l), app_\lambda(x.t', u, l)) \in R}$$

$$\frac{(u, u') \in R}{(app_\lambda(x.t, u, l), app_\lambda(x.t, u', l)) \in R} \qquad \frac{(l, l') \in R'}{(app_\lambda(x.t, u, l), app_\lambda(x.t, u, l')) \in R}$$

$$\frac{(u, u') \in R}{(app_v(x, u, l), app_v(x, u', l)) \in R} \qquad \frac{(l, l') \in R'}{(app_v(x, u, l), app_v(x, u, l')) \in R}$$

$$\frac{(u, u') \in R}{(u :: l, u' :: l) \in R'} \qquad \frac{(l, l') \in R'}{(u :: l, u :: l') \in R'}$$

Notation. *Again, we will be using the same notation for relations that was used in the previous chapters. The compatible closure of a binary relation on $\vec{\lambda}$-terms R is denoted as $\to_R$. The reflexive transitive closure of $\to_R$ is denoted as $\twoheadrightarrow_R$.*

Lemma 5 (Compatibility lemmas)**.** *Let R and R' be two binary relations on $\vec{\boldsymbol{\lambda}}$-terms and $\vec{\boldsymbol{\lambda}}$-lists respectively. If R and R' are compatible, then they satisfy:*

$$\frac{(l_1, l_1') \in R'}{(l_1 + l_2, l_1' + l_2) \in R'} \qquad \frac{(l_2, l_2') \in R'}{(l_1 + l_2, l_1 + l_2') \in R'}$$

$$\frac{(t, t') \in R}{(t@(u,l), t'@(u,l)) \in R} \qquad \frac{(u, u') \in R}{(t@(u,l), t@(u',l)) \in R} \qquad \frac{(l, l') \in R'}{(t@(u,l), t@(u,l')) \in R}$$

Proof. The proof proceeds easily by induction on lists for the append cases. For the compatibility cases of @ operation, proof follows by inspection of the principle argument and application of the append cases. □

We now make some considerations about β-normal forms in this system.

Definition 39 (β-normal form)**.** *We say that a $\vec{\boldsymbol{\lambda}}$-term t is in β-normal form when there exists no $\vec{\boldsymbol{\lambda}}$-term t' such that*

$$t \to_\beta t'.$$

Definition 40. *We inductively define the sets of $\vec{\boldsymbol{\lambda}}$-terms and $\vec{\boldsymbol{\lambda}}$-lists, respectively NT and NL, as follows:*

$$\frac{}{var(x) \in \mathit{NT}} \qquad \frac{t \in \mathit{NT}}{\lambda x.t \in \mathit{NT}} \qquad \frac{u \in \mathit{NT} \quad l \in \mathit{NL}}{app_v(x,u,l) \in \mathit{NT}} \qquad \frac{}{[] \in \mathit{NL}} \qquad \frac{u \in \mathit{NT} \quad l \in \mathit{NL}}{u :: l \in \mathit{NL}}.$$

Claim 4. *Given a $\vec{\boldsymbol{\lambda}}$-term t, the following are equivalent:*

(i) $t \in$ NT.

(ii) t is in β-normal form.

Remark. *One could simply describe the β-normal forms of $\vec{\boldsymbol{\lambda}}$ as the terms and lists with no occurrences of the constructor app_λ. This description is similar to idea of cut-elimination from sequent calculus (where the normal forms are the expressions not using cuts) and is one of the motivations for working with such systems. This system offers thus an advantage in comparison to the $\boldsymbol{\lambda}$-calculus, where a description of β-normal forms is more elaborated.*

We will not prove this claim here. However, we will come back to this claim in the next chapter to provide an alternative argument for the bijection between β-normal forms of $\boldsymbol{\lambda}$-terms and $\vec{\boldsymbol{\lambda}}$-terms.

We conclude the description of system $\vec{\boldsymbol{\lambda}}$ by presenting its typing system.

Definition 41 (Sequent)**.** *A sequent on terms $\Gamma \vdash t : A$ is a triple of a context, a $\vec{\boldsymbol{\lambda}}$-term and a simple type. A sequent on lists $\Gamma; A \vdash l : B$ is a quadruple of a context, a simple type, a $\vec{\boldsymbol{\lambda}}$-list and another simple type.*

Definition 42 (Typing rules for $\vec{\boldsymbol{\lambda}}$-expressions)**.**

$$\frac{}{x : A, \Gamma \vdash var(x) : A}\ \mathit{Var} \qquad \frac{x : A, \Gamma \vdash t : B}{\Gamma \vdash \lambda x.t : A \supset B}\ \mathit{Abs}$$

$$\frac{\Gamma, x : A \supset B \vdash u : A \qquad \Gamma, x : A \supset B; B \vdash l : C}{\Gamma, x : A \supset B \vdash app_v(x, u, l) : C}\ \mathit{App}_v$$

$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash u : A \qquad \Gamma; B \vdash l : C}{\Gamma \vdash app_\lambda(x.t, u, l) : C}\ \mathit{App}_\lambda$$

$$\frac{}{\Gamma; A \vdash [] : A}\ \mathit{Nil} \qquad \frac{\Gamma \vdash u : A \qquad \Gamma; B \vdash l : C}{\Gamma; A \supset B \vdash u :: l : C}\ \mathit{Cons}$$

4.2 $\vec{\lambda}$ vs the canonical subsystem of λm

In this section we prove an isomorphism between $\vec{\boldsymbol{\lambda}}$ and the canonical subsystem in $\boldsymbol{\lambda m}$. We start by defining two functions that play a key role in this isomorphism.

Definition 43. *Consider the following map i : $\vec{\boldsymbol{\lambda}}$-terms $\rightarrow Can$, recursively defined as follows:*

$$\begin{aligned} i(var(x)) &= x \\ i(\lambda x.t) &= \lambda x.i(t) \\ i(app_v(x, u, l)) &= x(i(u), i(l)) \\ i(app_\lambda(x.t, u, l)) &= (\lambda x.i(t))(i(u), i(l)) \\ i([]) &= [] \\ i(u :: l) &= i(u) :: i(l). \end{aligned}$$

Definition 44. *Consider the following map p : $\boldsymbol{\lambda m}$-terms $\rightarrow$ $\vec{\boldsymbol{\lambda}}$-terms, recursively defined as follows:*

$$\begin{aligned} p(x) &= var(x) \\ p(\lambda x.t) &= \lambda x.p(t) \\ p(t(u, l)) &= p(t)@(p(u), p(l)) \\ p([]) &= [] \\ p(u :: l) &= p(u) :: p(l). \end{aligned}$$

The following diagram summarizes the connection between the defined maps and map h defined in chapter 3.

$$\begin{tikzcd} & \boldsymbol{\lambda m} \arrow[dl, "p"'] \arrow[d, "h"] \\ \vec{\boldsymbol{\lambda}} \arrow[r, "i"'] & Can \end{tikzcd}$$

We begin by proving that the diagram shown is commutative. This will require the following auxiliary result.

Lemma 6. *Given* $\vec{\boldsymbol{\lambda}}$*-terms* t, u *and* $\vec{\boldsymbol{\lambda}}$*-list* l*,*

$$i(t@(u, l)) = app(i(t), i(u), i(l)).$$

Proof. The proof proceeds easily by inspection of the $\vec{\boldsymbol{\lambda}}$-term t. □

Theorem 2.

$$i \circ p = h$$

Proof. The equality is proved easily by induction on the structure of $\boldsymbol{\lambda m}$-expressions, using Lemma 6 in the application case. □

4.2.1 Bijection at the level of terms

Corollary 3.

$$i \circ p|_{Can} = id_{Can}$$

Proof. The equality is obtained via rewriting with Proposition 2 and then using Theorem 2. □

Theorem 3.

$$p \circ i = id_{\vec{\boldsymbol{\lambda}}\text{-terms}}$$

Proof. The proof proceeds easily by induction on the structure of the $\vec{\boldsymbol{\lambda}}$-expressions. □

4.2.2 Isomorphism at the level of reduction

In our subsytem of canonical terms, the substitution is not closed for the substitution operation. We have the following result that relates the two notions of substitution.

Lemma 7. *For every* $\vec{\boldsymbol{\lambda}}$*-term* t*,*

$$i(t[x := u]) = h(i(t)[x := i(u)]).$$

Proof. The proof proceeds by induction on the structure of the $\vec{\boldsymbol{\lambda}}$-term t.

For the case where $t = app_v(x, u, l)$, we use Lemma 6 to rewrite the term $i(t[x := v]) = i(v@(u, l))$ as $app(i(v), i(u), i(l))$. □

Lemma 8. *For every* $\boldsymbol{\lambda m}$*-terms* t*,*

$$p(t[x := u]) = p(t)[x := p(u)].$$

Proof. The proof proceeds easily by induction on the structure of the $\boldsymbol{\lambda m}$-term t. □

The following technical lemma says that we can derive the compatibilty rules of the system $\vec{\boldsymbol{\lambda}}$ given the canonoical closure of a compatible relation on $\boldsymbol{\lambda m}$.

Lemma 9. *Let* R *and* R' *be two binary relations on* $\boldsymbol{\lambda m}$*-terms and* $\boldsymbol{\lambda m}$*-lists respectively.*

The following binary relations are compatible in $\vec{\boldsymbol{\lambda}}$*:*

$$I = \{(t, t') \mid i(t) \ (\to_R)_c \ i(t'), \text{ for } \vec{\boldsymbol{\lambda}}\text{-terms } t, t'\}$$
$$I' = \{(l, l') \mid i(l) \ (\to_{R'})_c \ i(l'), \text{ for } \vec{\boldsymbol{\lambda}}\text{-lists } l, l'\}$$

Proof. We detail the proof of one of the compatibility cases:

$$\frac{(t, t') \in I}{(app_\lambda(x.t, u, l), app_\lambda(x.t', u, l)) \in I}.$$

From the definition of I, $(t, t') \in I \implies i(t) \ (\to_R)_c \ i(t')$.

Then, from the definition of the canonical closure relation, we have that there exist $\boldsymbol{\lambda m}$-terms t_1 and t_2 such that $h(t_1) = i(t)$ and $h(t_2) = i(t')$ and $t_1 \to_R t_2$.

We have:

$$\cfrac{\cfrac{\cfrac{t_1 \to_R t_2}{\lambda x.t_1 \to_R \lambda x.t_2}\ \text{(compatibility of } \to_R\text{)}}{(\lambda x.t_1)(i(u), i(l)) \to_R (\lambda x.t_2)(i(u), i(l))}\ \text{(compatibility of } \to_R\text{)}}{h((\lambda x.t_1)(i(u), i(l))) \ (\to_R)_c \ h((\lambda x.t_2)(i(u), i(l)))}\ \text{(canonical closure definition)}$$

Computing h, we get $(\lambda x.h(t_1))(h(i(u)), h'(i(l))) \ (\to_R)_c \ (\lambda x.h(t_2))(h(i(u)), h'(i(l)))$.

As $i(u) \in Can$, $h(i(u)) = i(u)$. And also, because $i(l) \in CanList$, we get that $h'(i(l)) = i(l)$. Hence,

$$(\lambda x.h(t_1))(i(u), i(l)) = (\lambda x.i(t))(i(u), i(l)) = i(app_\lambda(x.t, u, l))$$
$$(\to_R)_c \ (\lambda x.h(t_2))(i(u), i(l)) = (\lambda x.i(t'))(i(u), i(l)) = i(app_\lambda(x.t', u, l))$$

Therefore, by definition of I, we get that $(app_\lambda(x.t, u, l), app_\lambda(x.t', u, l)) \in I$. □

Theorem 4. *For every $\vec{\lambda}$-terms t, t',*

$$t \to_\beta t' \implies i(t)\ (\to_\beta)_c\ i(t').$$

Proof. The proof proceeds by induction on the relation $\to_\beta$ of $\vec{\lambda}$-expressions.

Lemma 7 deals with substitution preservation in the β-reduction cases.

Lemma 9 deals with all the compatibility cases. □

Theorem 5. *For every $t, t' \in Can$,*

$$t\ (\to_\beta)_c\ t' \implies p(t) \to_\beta p(t').$$

Proof. The proof starts by inverting our hypothesis $t\ (\to_\beta)_c\ t'$. The inversion provides that there exist λm-terms t_0, t_0' such that $t_0 \to_\beta t_0'$ and $t = h(t_0)$ and $t' = h(t_0')$. The proof then proceeds by induction on the relation step $t_0 \to_\beta t_0'$.

Lemma 8 deals with substitution preservation in the β-reduction cases.

Lemma 5 is useful in some compatibility cases. □

Of course that the preservation of reduction by map ψ follows trivially by the previous theorem, as ψ is a particular case of ψ'.

Summarising the previous results, we state the following corollary.

Corollary 4 (Isomorphism of reduction)**.**

1. $t \to_\beta t'$ *in* $\vec{\lambda} \iff i(t)\ (\to_\beta)_c\ i(t')$ *in* Can
2. $t\ (\to_\beta)_c\ t'$ *in* $Can \iff p(t) \to_\beta p(t')$ *in* $\vec{\lambda}$

Proof. Immediate by Theorems 3 to 5 and Corollary 3. □

4.2.3 Isomorphism at the level of typed terms

We start by establishing admissibility of the typing rules for the append and @ operations.

Lemma 10 (Append admissibility)**.** *The following rule is admissible in $\vec{\lambda}$:*

$$\frac{\Gamma; A \vdash l : B \qquad \Gamma; B \vdash l' : C}{\Gamma; A \vdash l + l' : C}.$$

Proof. The proof proceeds easily by induction on the list l. □

Lemma 11 (@ admissibility)**.** *The following rule is admissible in $\vec{\lambda}$:*

$$\frac{\Gamma \vdash t : A \supset B \qquad \Gamma \vdash u : A \qquad \Gamma; B \vdash l : C}{\Gamma \vdash t@(u, l) : C}.$$

Proof. The proof proceeds easily by inspection of t, using Lemma 10 when t is an application. ☐

We are now ready to prove the two theorems that provide the isomorphism of the canonical subsystem of $\boldsymbol{\lambda m}$ and system $\vec{\lambda}$ at the typing level.

Theorem 6 (i admissibility)**.** *For every $\vec{\lambda}$-term t and $\vec{\lambda}$-list l, the following rules hold:*

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash_c i(t) : A} \qquad\qquad \frac{\Gamma; A \vdash l : B}{\Gamma; A \vdash_c i(l) : B}.$$

Proof. The proof proceeds easily by simultaneous induction on the typing derivations of the premises. ☐

Theorem 7 (p admissibility)**.** *For every $t \in Can$ and $l \in CanList$, the following rules hold:*

$$\frac{\Gamma \vdash_c t : A}{\Gamma \vdash p(t) : A} \qquad\qquad \frac{\Gamma; A \vdash_c l : B}{\Gamma; A \vdash p(l) : B}.$$

Proof. From Proposition 2 we have that $h(t) = t$ and $h'(l) = l$. Then, inverting Definition 33, we have (in $\boldsymbol{\lambda m}$):

$$\Gamma \vdash t : A \qquad\qquad \Gamma; A \vdash l : B.$$

Thus, the proof proceeds easily by simultaneous induction on the above typing derivations of $\boldsymbol{\lambda m}$. Lemma 11 is crucial for the application case. ☐

Our argument for the isomorphism between the canonical subsystem in $\boldsymbol{\lambda m}$ and $\vec{\lambda}$ ends here. From now on, we will use the self contained representation, system $\vec{\lambda}$, to talk about canonical terms.

4.3 Conservativeness

The result of conservativeness establishes the connection between reduction in $\vec{\lambda}$ and in $\boldsymbol{\lambda m}$. We start by proving a auxiliary result that connects the @ operation with list append.

Lemma 12. *For every $\vec{\lambda}$-terms t, u, $\boldsymbol{\lambda m}$-term v and $\boldsymbol{\lambda m}$-lists l, l', the following equality holds.*

$$(t@(u, p(l)))@(p(v), p(l')) = t@(u, p(l + (v :: l')))$$

Proof. The proof proceeds by inspection of $\vec{\lambda}$-term t and uses the simple fact that $p(l + (v :: l')) = p(l) + p(v) :: p(l')$. ☐

Theorem 8 (Conservativeness)**.** *For every $\vec{\boldsymbol{\lambda}}$-terms t and t', we have:*

$$t \twoheadrightarrow_{\beta} t' \iff i(t) \twoheadrightarrow_{\beta h} i(t').$$

Proof. $\boxed{\Longrightarrow}$ Let t and t' be $\vec{\boldsymbol{\lambda}}$-terms.

For this implication it suffices to mimic β steps of the system $\vec{\boldsymbol{\lambda}}$ in the system $\boldsymbol{\lambda m}$.

Case $t \rightarrow_{\beta_1} t'$:

$$\begin{tikzcd}
\text{(in } \vec{\boldsymbol{\lambda}}\text{)} & \text{(in } \boldsymbol{\lambda m}\text{)} & \\
& (\lambda x.i(t))(i(u),[]) \arrow[d, "\beta_1"'] & \\
app_\lambda(x.t,u,[]) \arrow[ur, mapsto, "i"] \arrow[dd, "\beta_1"'] & i(t)[x:=i(u)] \arrow[d, two heads, "h"'] & \\
& h(i(t)[x:=i(u)]) \arrow[d, equal] & \text{Proposition 1} \\
t[x:=u] \arrow[r, mapsto, "i"] & i(t[x:=u]) & \text{Lemma 7}
\end{tikzcd}$$

Case $t \rightarrow_{\beta_2} t'$:

$$\begin{tikzcd}
\text{(in } \vec{\boldsymbol{\lambda}}\text{)} & \text{(in } \boldsymbol{\lambda m}\text{)} & \\
& (\lambda x.i(t))(i(u),i(v)::i(l)) \arrow[d, "\beta_2"] & \\
app_\lambda(x.t,u,v::l) \arrow[ur, mapsto, "i"] \arrow[dddd, "\beta_2"] & (i(t)[x:=i(u)])(i(v),i(l)) \arrow[d, two heads, "h"] & \\
& h(i(t)[x:=i(u)])(i(v),i(l)) \arrow[d, equal] & \text{Proposition 1} \\
& (i(t[x:=u]))(i(v),i(l)) \arrow[d, two heads, "h"] & \text{Lemma 7} \\
& app(i(t[x:=u]),i(v),i(l)) \arrow[d, equal] & \text{Lemma 3} \\
t[x:=u]@(v,l) \arrow[r, mapsto, "i"] & i(t[x:=u]@(v,l)) & \text{Lemma 6}
\end{tikzcd}$$

$\boxed{\Longleftarrow}$ Let t and t' be $\boldsymbol{\lambda m}$-terms.

For this implication, we first show how a reduction $t \to_{\beta h} t'$ in $\boldsymbol{\lambda m}$ is directly translated into a reduction $p(t) \to_{\beta} p(t')$ in $\vec{\boldsymbol{\lambda}}$.

Case $t \to_{\beta_1} t'$:

$$\begin{tikzcd}
\text{(in } \boldsymbol{\lambda m}\text{)} & \text{(in } \vec{\boldsymbol{\lambda}}\text{)} & \\
& (\lambda x.p(t))@(p(u),[]) \arrow[d, equal] & \text{Def. of @} \\
(\lambda x.t)(u,[]) \arrow[ur, mapsto, "p"] \arrow[d, "\beta_1"] & app_{\lambda}(x.p(t),p(u),[]) \arrow[d, "\beta_1"] & \\
t[x:=u] \arrow[dr, mapsto, "p"] & p(t)[x:=p(u)] \arrow[d, equal] & \text{Lemma 8} \\
& p(t[x:=u]) &
\end{tikzcd}$$

Case $t \to_{\beta_2} t'$:

$$\begin{tikzcd}
\text{(in } \boldsymbol{\lambda m}\text{)} & \text{(in } \vec{\boldsymbol{\lambda}}\text{)} & \\
& (\lambda x.p(t))@(p(u),p(v)::p(l)) \arrow[d, equal] & \text{Def. of @} \\
(\lambda x.t)(u,v::l) \arrow[ur, mapsto, "p"] \arrow[d, "\beta_2"] & app_{\lambda}(x.p(t),p(u),p(v)::p(l)) \arrow[d, "\beta_2"] & \\
t[x:=u](v,l) \arrow[dr, mapsto, "p"] & p(t)[x:=p(u)]@(p(v),p(l)) \arrow[d, equal] & \text{Lemma 8} \\
& p(t[x:=u])@(p(v),p(l)) &
\end{tikzcd}$$

Case $t \to_{h} t'$:

$$\begin{tikzcd}
\text{(in } \boldsymbol{\lambda m}\text{)} & \text{(in } \vec{\boldsymbol{\lambda}}\text{)} & \\
t(u,l)(u',l') \arrow[r, mapsto, "p"] \arrow[d, "h"] & (p(t)@(p(u),p(l)))@(p(u'),p(l')) \arrow[d, equal] & \text{Lemma 12} \\
t(u,l+(u'::l')) \arrow[r, mapsto, "p"] & p(t)@(p(u),p(l+(u'::l'))) &
\end{tikzcd}$$

From the shown base cases, an easy induction on the relation $\to_{\beta h}$ proves for every $\boldsymbol{\lambda m}$-terms t, t':

$$t \twoheadrightarrow_{\beta h} t' \implies p(t) \twoheadrightarrow_{\beta} p(t').$$

Thus, for every $\vec{\boldsymbol{\lambda}}$-terms u, u',

$$i(u) \twoheadrightarrow_{\beta h} i(u') \implies p(i(u)) = u \twoheadrightarrow_{\beta} p(i(u')) = u'.$$

$\square$

As an immediate application of the conservativeness result just proved, we can derive subject reduction for $\vec{\boldsymbol{\lambda}}$ from $\boldsymbol{\lambda m}$ (already proved as Theorem 1).

Corollary 5 (Subject Reduction for $\vec{\boldsymbol{\lambda}}$)**.** *Given $\vec{\boldsymbol{\lambda}}$-terms t and t', the follwing holds:*

$$\Gamma \vdash t : A \wedge t \to_{\beta} t' \implies \Gamma \vdash t' : A.$$

Proof. The proof is shown in a derivation-like style. We use dashed lines for derivations that do not follow from typing rules or rules already proven admissible.

$$
\text{Theorem 3}\ \dfrac{\text{Theorem 7}\ \dfrac{\text{Proposition 2}\ \dfrac{\text{Definition 33}\ \dfrac{\text{Corollary 1}\ \dfrac{\text{Corollary 1}\ \dfrac{\text{*Definition 33}\ \dfrac{\text{Theorem 6}\ \dfrac{\Gamma \vdash t : A}{\Gamma \vdash_c i(t) : A}}{\Gamma \vdash t_0 : A} \qquad {\twoheadrightarrow_h} \subset {\twoheadrightarrow_{\beta h}}\ \dfrac{\text{Proposition 1}\ \dfrac{}{t_0 \twoheadrightarrow_h h(t_0)}}{t_0 \twoheadrightarrow_{\beta h} h(t_0)}}{\Gamma \vdash i(t) : A} \qquad \text{Theorem 8}\ \dfrac{t \to_{\beta} t'}{i(t) \twoheadrightarrow_{\beta h} i(t')}}{\Gamma \vdash i(t') : A}}{\Gamma \vdash_c h(i(t')) : A}}{\Gamma \vdash_c i(t') : A}}{\Gamma \vdash p(i(t')) : A}}{\Gamma \vdash t' : A}
$$

*Definition 33: inverting this definition with $\Gamma \vdash_c i(t) : A$ we get that there exists a $\boldsymbol{\lambda m}$-term t_0 such that $h(t_0) = i(t)$ and that $\Gamma \vdash t_0 : A$. $\square$

As another consequence of conservativeness, confluence of $\vec{\boldsymbol{\lambda}}$ can be lifted to $\boldsymbol{\lambda m}$.

Corollary 6 (Confluence lift)**.** *If $\twoheadrightarrow_{\beta}$ is confluent in $\vec{\boldsymbol{\lambda}}$, then $\twoheadrightarrow_{\beta h}$ is confluent in $\boldsymbol{\lambda m}$.*

Proof. Given $\boldsymbol{\lambda m}$-terms t, t_1, t_2, we have the following:

$$\begin{tikzcd}
 & & t \arrow[dll, two heads, "\beta h"'] \arrow[d, "h"] \arrow[drr, two heads, "\beta h"] & & \\
t_1 \arrow[d, "h"] & & i(p(t)) = h(t) \arrow[dll, two heads, "\beta h"'] \arrow[drr, two heads, "\beta h"] \arrow[dd, dashed, "\beta h"] & & t_2 \arrow[d, "h"] \\
i(p(t_1)) = h(t_1) \arrow[drr, two heads, "\beta h"] & & & & i(p(t_2)) = h(t_2) \arrow[dll, two heads, "\beta h"] \\
 & & \exists\, i(t') & &
\end{tikzcd}$$

In the latter diagram, the existance of such t' comes from the fact that $\vec{\boldsymbol{\lambda}}$ is confluent and from using Theorem 8 to lift the following diagram to $\boldsymbol{\lambda m}$.

$$\begin{tikzcd}
 & p(t) \arrow[dl, two heads, "\beta"] \arrow[dr, two heads, "\beta"] \arrow[dd, dashed, "\beta"] & \\
p(t_1) \arrow[dr, two heads, "\beta"] & & p(t_2) \arrow[dl, two heads, "\beta"] \\
 & \exists\, t' &
\end{tikzcd}$$

$\square$

4.4 Mechanisation in Rocq

The mechanisation for the system $\vec{\boldsymbol{\lambda}}$ also uses the *Autosubst* library, and follows the same style of the mechanisation of the system $\boldsymbol{\lambda m}$, except for the nonstandard substitution operation (that we cover in more detail by the end of the chapter).

4.4.1 `Canonical.v`

Most definitions for the canonical self-contained subsystem follow from the definitions for the system $\boldsymbol{\lambda m}$ with small adaptations. In particular, this applies to the definitions of terms, lists, reduction and the typing relation.

```
(* syntax *)
Inductive term: Type :=
| Vari (x: var)
| Lamb (t: {bind term})
```

```
| VariApp (x: var) (u: term) (l: list term)
| LambApp (t: {bind term}) (u: term) (l: list term).
...
(* reduction relations *)
Inductive β1: relation term :=
| Step_Beta1 (t: {bind term}) (t' u: term) :
  t' = t.[u .: ids] → β1 (LambApp t u []) t'.

Inductive β2: relation term :=
| Step_Beta2 (t: {bind term}) (t' u v: term) l :
  t' = t.[u .: ids]@(v,l) → β2 (LambApp t u (v::l)) t'.

Definition step := comp (union _ β1 β2).
Definition step' := comp' (union _ β1 β2).

Definition multistep := clos_refl_trans_1n _ step.
Definition multistep' := clos_refl_trans_1n _ step'.
...
(* typing rules *)
Inductive sequent (Γ: var→type) : term → type → Prop :=
| varAxiom (x: var) (A: type) :
  Γ x = A → sequent Γ (Vari x) A
| Right (t: term) (A B: type) :
  sequent (A .: Γ) t B → sequent Γ (Lamb t) (Arr A B)
| Left (x: var) (u: term) (l: list term) (A B C: type) :
  Γ x = (Arr A B) → sequent Γ u A → list_sequent Γ B l C →
  sequent Γ (VariApp x u l) C
| KeyCut (t: {bind term}) (u: term) (l: list term) (A B C: type) :
  sequent (A .: Γ) t B → sequent Γ u A → list_sequent Γ B l C →
  sequent Γ (LambApp t u l) C
with list_sequent (Γ:var→type) : type → (list term) → type → Prop :=
| nilAxiom (C: type) : list_sequent Γ C [] C
| Lft (u: term) (l: list term) (A B C:type) :
  sequent Γ u A → list_sequent Γ B l C →
  list_sequent Γ (Arr A B) (u :: l) C.
```

The formalisation of the step relations works as shown for the system λm using a `comp` meta-relation for compatibility closure. In the next subsection we describe in more detail the aproach used to define the substitution operation for this system.

This module also contains proofs for every compatibility lemma (recall Lemma 5).

```
Section CompatibilityLemmas.
  Lemma step_comp_append1 :
    ∀l1 l1', step' l1 l1' → ∀l2, step' (l1 ++ l2) (l1' ++ l2).
  Proof.
    intros l1 l1' H.
    induction H ; intros.
    - repeat rewrite<- app_comm_cons.
      now constructor.
    - repeat rewrite<- app_comm_cons.
      constructor. now apply IHcomp'.
  Qed.
  ...
  Lemma step_comp_app2 :
    ∀v u u' l, step u u' → step v@(u,l) v@(u',l).
  ...
End CompatibilityLemmas.
```

4.4.2 `CanonicalIsomorphism.v`

This module contains every proof related to the isomorphism of the canonical subsystem in λm and the system $\vec{\lambda}$.

Let us see the statement of Lemma 9:

```
Lemma step_can_is_compatible :
  Canonical.is_compatible
    (fun t t' ⇒ step_can (i t) (i t'))
    (fun l l' ⇒ step_can' (map i l) (map i l')).
Proof.
  split ; intros ; asimpl ; inversion H.
  ...
```

We prove every compatibility step by inverting first the definition of `step_can`. Despite being a bureocratic result, it helps simplifying further proofs (such as Theorem 4 shown below) and reveals some benefits of formalising the general predicate of compatibility `is_compatible`.

```
Theorem i_step_pres :
  (∀(t t': Canonical.term),
      Canonical.step t t' → step_can (i t) (i t'))
  ∧
  (∀(l l': list Canonical.term),
      Canonical.step' l l' →
      step_can' (map i l) (map i l')).
Proof.
  pose step_can_is_compatible as Hic.
  destruct Hic.
  apply Canonical.mut_comp_ind ; intros ; auto.
  ...
```

The mechanised proof shown above makes use of the automation provided by the `auto` tactic by strategically adding relevant lemmas to the proof context. More specifically, the line `pose step_can_is_compatible as` adds to the proof context the fact that `step_can` is a compatible relation for $\vec{\lambda}$-terms.

4.4.3 Conservativeness.v

This module is only about the proof for the conservativeness theorem. The mechanised theorem follows exactly the proof given diagramatically in Theorem 8, divided into two parts, `conservativeness1` and `conservativeness2`, for each of the two concerned implications.

```
Theorem conservativeness :
  ∀t t', Canonical.multistep t t' ↔ LambdaM.multistep (i t) (i t').
Proof.
  split.
  - intro H.
    induction H as [| t1 t2 t3].
    + constructor.
    + apply multistep_trans with (i t2) ; try easy.
      * now apply conservativeness1.
  - intro H.
    rewrite<- (proj1 inversion2) with t.
    rewrite<- (proj1 inversion2) with t'.
    induction H as [| t1 t2 t3].
```

```
    + constructor.
    + apply multistep_trans with (p t2) ; try easy.
      * now apply conservativeness2.
Qed.
```

4.4.4 A closer look at the mechanisation

a) Autosubst and a nonstandard substitution operation

One of the most peculiar definitions in system $\vec{\lambda}$ is the substitution operation (Definition 36). As referred before, we have an unusual behaviour for the constructor app_v. In practice, on a substitution $app_v(x, u, l)[x := t]$, there occurs an inspection of the term t that dictates the result of the substitution operation.

As we are working with the *Autosubst* library, we tried to automatically generate the substitution operation for our case. But as expected, the `derive` tactic failed to give us the desired operation:

```
Subst_term =
(fix dummy (σ : var → term) (s : term) {struct s} : term :=
match s as t return (annot term t) with
   | Vari x ⇒ (fun x0: var ⇒ σ x0) x
   | Lamb t ⇒ (fun t0: {bind term} ⇒ Lamb t0.[up σ]) t
   | VariApp x u l ⇒ (fun (x0: var) (_: term) (_: list term) ⇒ σ x0) x u l
   | LambApp t u l ⇒
       (fun (t0: {bind term}) (s0: term) (l0: list term) ⇒
        LambApp t0.[up σ] s0.[σ] l0..[σ]) t u l
   end)
```

Therefore, we gave the proof assistant our dedicated definition (directly as a proof object, as seen below).

```
Definition app (t u: term) (l: list term): term :=
  match t with
  | Vari x ⇒ VariApp x u l
  | Lamb t' ⇒ LambApp t' u l
  | VariApp x u' l' ⇒ VariApp x u' (l' ++ u::l)
  | LambApp t' u' l' ⇒ LambApp t' u' (l' ++ u::l)
  end.
Notation "t '@(' u ',' l ')'" := (app t u l) (at level 9).
```

```
...
Instance Ids_term : Ids term. derive. Defined.
Instance Rename_term : Rename term. derive. Defined.
Instance Subst_term : Subst term.
Proof.
  unfold Subst. fix inst 2. change _ with (Subst term) in inst.
  intros σ s. change (annot term s). destruct s.
  - exact (σ x).
  - exact (Lamb (subst (up σ) t)).
  - exact ((σ x)@(subst  s, mmap (subst σ) l)).
  - exact (LambApp (subst (up σ) t) (subst σ s) (mmap (subst σ) l)).
Defined.
```

The downside to our approach was the need to manually prove every substitution lemma required by the *Autosubst* instance `SubstLemmas`. However, proving such lemmas was crucial to enjoy the automation provided from the library for the mechanised inductive type of the $\vec{\lambda}$-terms.

Chapter 5

The isomorphism $\lambda \cong \vec{\lambda}$

In chapter 2, the simply typed $\boldsymbol{\lambda}$-calculus was introduced. Now, we show an isomorphism between the system $\vec{\boldsymbol{\lambda}}$ introduced in the previous chapter and the simply typed $\boldsymbol{\lambda}$-calculus. This isomorphism will come at the level of terms, reduction, β-normal forms and typing rules.

This isomorphism is of great interest as $\vec{\boldsymbol{\lambda}}$ typing rules resemble a sequent calculus style. In this sense, the isomorphism studided in this chapter establishes a correspondence between natural deduction system (the simply typed $\boldsymbol{\lambda}$-calculus) and a fragment of sequent calculus (the system $\vec{\boldsymbol{\lambda}}$). The chapter is inspired in the works [12] and [13, Chapter 4] and is summarized by the following diagram:

$$\begin{tikzcd}
\boxed{\lambda} \arrow[r, shift left, "\psi"] \arrow[d, "\downarrow_\beta"] & \boxed{\vec{\lambda}} \arrow[l, shift left, "\theta"] \arrow[d, "\downarrow_\beta"] \\
\boxed{\text{NF}} \arrow[r, leftrightarrow, "\cong"] & \boxed{\text{NT}}
\end{tikzcd}$$

In this diagram: the horizontal arrows symbolize the inverse maps underlying the isomorphism; the down arrows symbolize (partial) maps associating expressions to the respective beta-normal form (when existing). Also recall the sets NF$\subset$ $\boldsymbol{\lambda}$-terms and NT$\subset$ $\vec{\boldsymbol{\lambda}}$-terms of the respective β-normal forms.

5.1 Mappings θ and ψ

We start by defining the maps between expressions of $\boldsymbol{\lambda}$ and $\vec{\boldsymbol{\lambda}}$ underlying the isomorphism.

Definition 45 (Maps θ and θ')**.** *The map θ : $\vec{\boldsymbol{\lambda}}$-terms $\rightarrow$ $\boldsymbol{\lambda}$-terms is defined simultaneously with the map θ' : ($\boldsymbol{\lambda}$-terms $\times$ $\vec{\boldsymbol{\lambda}}$-lists) $\rightarrow$ $\boldsymbol{\lambda}$-terms by recursion on $\vec{\boldsymbol{\lambda}}$-terms and $\vec{\boldsymbol{\lambda}}$-lists respectively, as follows:*

$$\begin{aligned}
\theta(var(x)) &= x \\
\theta(\lambda x.t) &= \lambda x.\theta(t) & \theta'(M, []) &= M \\
\theta(app_v(x, u, l)) &= \theta'(x, u :: l) & \theta'(M, u :: l) &= \theta'(M\ \theta(u), l). \\
\theta(app_\lambda(x.t, u, l)) &= \theta'(\lambda x.\theta(t), u :: l)
\end{aligned}$$

Definition 46. *Maps ψ and ψ' The map ψ' : ($\boldsymbol{\lambda}$-terms $\times$ $\vec{\boldsymbol{\lambda}}$-lists) $\rightarrow$ $\vec{\boldsymbol{\lambda}}$-terms is defined by recursion on*

$\boldsymbol{\lambda}$-terms as follows:

$$\begin{aligned}
\psi(x, []) &= var(x) \\
\psi(x, u :: l) &= app_v(x, u, l) \\
\psi(\lambda x.M, []) &= \lambda x.\psi(M) \\
\psi(\lambda x.M, u :: l) &= app_\lambda(x.\psi(M), u, l) \\
\psi(MN, l) &= \psi'(M, \psi(N) :: l),
\end{aligned}$$

where $\psi(M)$ is easily defined as $\psi'(M, [])$.

5.1.1 Bijection at the level of terms

Now, we will establish that θ and ψ are indeed inverse maps, and thus, $\boldsymbol{\lambda}$-terms and $\vec{\boldsymbol{\lambda}}$-terms are in bijection.

Lemma 13.

$$\theta \circ \psi' = \theta'$$

Proof. The proof proceeds by induction on the structure of $\boldsymbol{\lambda}$-terms and proper inspection of the $\vec{\boldsymbol{\lambda}}$-list in the variable and abstraction cases. □

Theorem 9 (θ is left inverse of ψ)**.**

$$\theta \circ \psi = id_{\boldsymbol{\lambda}\text{-terms}}$$

Proof. Immediate using Lemma 13. □

Theorem 10 (ψ is left inverse of θ)**.**

$$\begin{aligned}
\psi \circ \theta &= id_{\vec{\boldsymbol{\lambda}}\text{-terms}} \\
\psi \circ \theta' &= \psi'
\end{aligned}$$

Proof. The proof proceeds by simultaneous induction on the structure of $\vec{\boldsymbol{\lambda}}$-terms and $\vec{\boldsymbol{\lambda}}$-lists, respectively. □

5.1.2 Isomorphism at the level of reduction

Now we turn our attention to reduction, showing that the reduction relations $\rightarrow_\beta$ of $\boldsymbol{\lambda}$-calculus and $\vec{\boldsymbol{\lambda}}$ are isomorphic.

First, introduce some lemmata, in order to relate the mappings θ' and ψ' with the @ operation and list append.

Lemma 14. *For every $\vec{\boldsymbol{\lambda}}$-terms t, u and $\vec{\boldsymbol{\lambda}}$-list l,*

$$\theta(t@(u, l)) = \theta'(\theta(t)\ \theta(u), l)$$

and also, for every $\boldsymbol{\lambda}$-term M, $\vec{\boldsymbol{\lambda}}$-term u' and $\vec{\boldsymbol{\lambda}}$-lists l, l',

$$\theta'(M, l + (u' :: l')) = \theta'(\theta'(M, l)\ \theta(u'), l').$$

Proof. The proof proceeds easily by simultaneous induction on the structure of the $\vec{\boldsymbol{\lambda}}$-term t and $\vec{\boldsymbol{\lambda}}$-list l, respectively. □

Corollary 7. *For every $\boldsymbol{\lambda}$-term M, $\vec{\boldsymbol{\lambda}}$-term u and $\vec{\boldsymbol{\lambda}}$-list l,*

$$\psi'(M, u :: l) = \psi(M)@(u, l).$$

Proof. The result follows as a corollary of Lemma 14, using Theorem 10 and Lemma 13 to rewrite the left-hand side of the equality. □

Using the previous lemmas, the preservation of the substitution operations by θ and ψ follow.

Lemma 15. *For every $\vec{\boldsymbol{\lambda}}$-terms t, u,*

$$\theta(t[x := u]) = \theta(t)[x := \theta(u)]$$

and also, for every $\boldsymbol{\lambda}$-term M, $\vec{\boldsymbol{\lambda}}$-term u and $\vec{\boldsymbol{\lambda}}$-list l,

$$\theta'(M[x := \theta(u)], l[x := u]) = \theta'(M, l)[x := u].$$

Proof. The proof follows by simultaneous induction on the structure of t and l, using Lemma 14. □

Lemma 16. *For every $\boldsymbol{\lambda}$-terms M, N and $\vec{\boldsymbol{\lambda}}$-list l,*

$$\psi'(M[x := N], l[x := \psi(N)]) = \psi'(M, l)[x := \psi(N)].$$

Proof. The proof follows by induction on the structure of $\boldsymbol{\lambda}$-term M, using Corollary 7. □

Now, we are essentially ready to obtain the isomorphism at the level of reduction.

Lemma 17. *For every $\boldsymbol{\lambda}$-terms M, N and $\vec{\boldsymbol{\lambda}}$-list l,*

$$M \to_\beta N \implies \theta'(M, l) \to_\beta \theta'(N, l).$$

Proof. The proof follows easily by induction on the structure of the $\vec{\boldsymbol{\lambda}}$-list l. □

Theorem 11 (Preservation of reduction by θ)**.** *For every $\vec{\boldsymbol{\lambda}}$-terms t, t',*

$$t \rightarrow_\beta t' \implies \theta(t) \rightarrow_\beta \theta(t')$$

and also, for every $\boldsymbol{\lambda}$-term M and $\vec{\boldsymbol{\lambda}}$-lists l, l',

$$l \rightarrow_\beta l' \implies \theta'(M, l) \rightarrow_\beta \theta(M, l').$$

Proof. The proof proceeds by simultaneous induction on the structure of the step relation on $\vec{\boldsymbol{\lambda}}$-expressions.

Lemma 14 is useful for the cases of compatibility steps.

Lemma 15 is crucial for cases dealing with β steps. □

Theorem 12 (Preservation of reduction by ψ')**.** *For every $\boldsymbol{\lambda}$-terms M, N and $\vec{\boldsymbol{\lambda}}$-list l,*

$$M \rightarrow_\beta N \implies \psi'(M, l) \rightarrow_\beta \psi'(N, l).$$

Proof. The proof proceeds by induction on the structure of the step relation on $\boldsymbol{\lambda}$-terms.

Lemma 16 is crucial for cases dealing with β steps. □

Of course that the preservation of reduction by map ψ follows trivially by the previous theorem, as ψ is a particular case of ψ'.

Summarising our isomorphism at the level of reduction, we state the following corollary.

Corollary 8 (Isomorphism of reduction)**.**

1. $t \rightarrow_\beta t'$ *in* $\vec{\boldsymbol{\lambda}} \iff \theta(t) \rightarrow_\beta \theta(t')$ *in* $\boldsymbol{\lambda}$
2. $M \rightarrow_\beta N$ *in* $\boldsymbol{\lambda} \iff \psi(M) \rightarrow_\beta \psi(N)$ *in* $\vec{\boldsymbol{\lambda}}$

Proof. Immediate by Theorems 9 to 12. □

5.1.3 Isomorphism at the level of β-normal forms

Now, we will argue that the bijection between $\boldsymbol{\lambda}$-terms and $\vec{\boldsymbol{\lambda}}$-terms still holds when we restrict to β-normal forms. For this, is convenient to recall both Definition 10 and Definition 40.

Theorem 13 (θ preserves β-nfs)**.**

$$t \in \mathit{NT} \implies \theta(t) \in \mathit{NF}$$

Proof. Given $t \in$ NT, by Claim 4, there exists no t' such that $t \to_\beta t'$ (or t is a β-nf).

Now let us prove that $\theta(t)$ is a β-nf.

Suppose there exists a $\boldsymbol{\lambda}$-term N such that $\theta(t) \to_\beta N$. From Theorem 12, it is also true that $\psi(\theta(t)) \to_\beta \psi(N)$. And, using Theorem 10, we may rewrite $\psi(\theta(t))$ as t, obtaining a contradiction as $t \to_\beta \psi(N)$ while t is a β-nf.

Therefore, such N cannot exist and $\theta(t)$ is a β-nf (consequently, from Claim 1, $\theta(t) \in$ NF). □

We can prove an analogous result for ψ.

Theorem 14 (ψ preserves β-nfs)**.**

$$M \in \mathit{NF} \implies \psi(M) \in \mathit{NT}$$

Proof. Analogous to the proof of Theorem 13. □

5.1.4 Isomorphism at the level of typed terms

Finally, we complete the proof of our isomorphism by showing a 1-1 correspondence between typed terms of the simply typed $\boldsymbol{\lambda}$-calculus and system $\vec{\boldsymbol{\lambda}}$.

Theorem 15 (θ admissibility)**.** *The following rules hold:*

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \theta(t) : A} \qquad \frac{\Gamma \vdash M : A \quad \Gamma; A \vdash l : B}{\Gamma \vdash \theta'(M, l) : B}$$

Proof. The proof proceeds easily by simultaneous induction on the structure of the typing derivations in the system $\vec{\boldsymbol{\lambda}}$. □

Theorem 16 (ψ' admissibility)**.** *The following rule holds:*

$$\frac{\Gamma \vdash M : A \quad \Gamma; A \vdash l : B}{\Gamma \vdash \psi'(M, l) : B}$$

Proof. The proof proceeds easily by induction on the structure of the typing derivations in the simply typed $\boldsymbol{\lambda}$-calculus. □

5.2 Mechanisation in Rocq

In this section we provide a brief description of the mechanisation of the concepts and results described in the previous section. Essentially, we just defined maps θ and ψ and mechanised each of the results provided.

One detail that should be highlighted is the definition for maps θ and θ'.

```
Fixpoint θ (t: Canonical.term) : Lambda.term :=
  match t with
  | Vari x ⇒ Var x
  | Lamb t ⇒ Lam (θ t)
  | VariApp x u l ⇒ fold_left (fun s v ⇒ App s (θ v)) (u::l) (Var x)
  | LambApp t u l ⇒ fold_left (fun s v ⇒ App s (θ v)) (u::l) (Lam (θ t))
  end.

Definition θ' (s: Lambda.term) (l: list Canonical.term) :
  Lambda.term := fold_left (fun s v ⇒ App s (θ v)) l s.
```

The mechanised object that represents map θ' uses a higher order function on lists called `fold_left` that behaves exactly as θ', given the function `(fun s v ⇒ App s (θ v))` which folds the $\vec{\boldsymbol{\lambda}}$-list into a $\boldsymbol{\lambda}$-term.

Such representation of map θ was an undesired consequence of the use of polymorphic lists in the definition for $\vec{\boldsymbol{\lambda}}$-terms. We could not define mutually recursive functions on the structure of the term and list because the proof assistant fails to recognise their termination [19]. Instead, we have to define these maps using higher order functions. In this specific case, we could even enjoy the generality of the `fold_left` function. Unfortunately, with such definition for θ', we had to repeatedly fold the definition for θ' in our proofs to make the goal more readable (hiding the calls to the `fold_left`). This necessity of "folding" a definition can be seen in the mechanisation of Lemma 17:

```
Lemma θ'_step_pres l :
  ∀s s', Lambda.step s s' → Lambda.step (θ' s l) (θ' s' l).
Proof.
  induction l as [| u l]; intros ; asimpl ; try easy.
  - fold (θ' (App s (θ u)) l).
    fold (θ' (App s' (θ u)) l).
    apply IHl. now constructor.
Qed.
```

Chapter 6

Conclusions

In this chapter, we first describe our contributions and then discuss possible directions for future work.

6.1 Contributions

We list below the contributions achieved with this dissertation.

First and more important, we used the *Rocq Prover* and the *Autosubst* library to mechanise the following systems introduced in this work:

1. the multiary $\boldsymbol{\lambda}$-calculus (system $\boldsymbol{\lambda m}$);
2. the canonical subsystem of $\boldsymbol{\lambda m}$;
3. the canonical $\boldsymbol{\lambda}$-calculus (system $\vec{\boldsymbol{\lambda}}$).

Then, using with the formalisation of these systems on the proof assistant, we also obtained computer-verified proofs for results such as:

1. subject reduction for systems $\boldsymbol{\lambda m}$ and $\vec{\boldsymbol{\lambda}}$;
2. isomorphism between the canonical subsystem of $\boldsymbol{\lambda m}$ and system $\vec{\boldsymbol{\lambda}}$;
3. conservativeness of $\vec{\boldsymbol{\lambda}}$ over $\boldsymbol{\lambda m}$;
4. isomorphism between the simply typed $\boldsymbol{\lambda}$-calculus and system $\vec{\boldsymbol{\lambda}}$;
5. confluence of systems $\vec{\boldsymbol{\lambda}}$ and $\boldsymbol{\lambda m}$.

Second, we gave an exhaustive definition for the concept of subsystem, separating two isomorphic representations of the canonical subsystem of $\boldsymbol{\lambda m}$. This helped us clarify the loose idea of subsystem and simplify some of the result proven (for example, using the self-contained system $\vec{\boldsymbol{\lambda}}$ for proving the theorem of conservativeness). From this idea, we could even propose a standard approach to formalise any subsystem.

Third and last, through this document, a detailed exposition of the mechanised systems and proofs using the *Rocq Prover* along with some digressions over our formalisation choices.

6.2 Discussion and related work

Now, we provide some discussion over our techniques for the mechanisation of metatheory, connecting them with related work.

6.2.1 De Bruijn indices

As introduced earlier, de Bruijn indices (introduced in [9]) is a technique to define a capture-avoiding substitution by working with expressions up to α-equivalence. In the original work of de Bruijn, the use of parallel substitutions is already present, as a way to simplify the presentation of the substitution operation and at the same time generalising it. This technique is often criticised for its unreadability and distance to the systems and results written in paper.

The literature is vast on other alternatives for representing syntax with binders [5, Section 2.3]. Therefore, one can find a formalisation for the $\boldsymbol{\lambda}$-calculus in these many flavours: nominal [29], locally nameless [23] and HOAS [10], to name a few. We chose to use de Bruijn syntax in the proof assistant as, in our case, it was a way to avoid digressions over metatheory that is not central to our objectives.

6.2.2 Autosubst library

The *Autosubst* library [26] was indeed a central choice along our work of formalisation using the *Rocq Prover*. It is an accessible tool for the mechanisation of metatheory of general syntax with binders that relies on the use of parallel substitutions and σ-calculus theory in order to simplify and automatise the metatheory around substitution operations. Moreover, many of the operations provided for substitutions can work when using a general concept of typing systems with infinite contexts.

Other well-known libraries/code generators for mechanising syntax with binders in the *Rocq Prover* are *GMeta* [21] (a code generator for generic representations), *Dblib* [24] (a library for representations using de Bruijn indices) and *LNGen* [4] (a code generator for locally nameless representations). However, these libraries do not have support for expressions with many syntactical classes and have a higher cost of entrance (in our opinion) comparing with *Autosubst*.

Another code generator that should be highlighted is *Autosubst2* [27]. Using the *Autosubst* library seems an unusual choice, considering the existence of a code generator that appeared to fix many of its known problems, like not supporting many-sorted syntaxes. Adding to this, many use cases prove its effectiveness in the mechanisation of metatheory [15, 11, 2]. We can argue for the use of the less sophisticated *Autosubst* library in two ways. First, we were able to achieve the desired support in the case of our systems thanks to the use of polymorphic lists. Second, in the case of our system with a non conventional substitution operation, we would have nothing generated by *Autosubst2*, as this system has an unexpected behaviour. Using a library tool instead of a code generator allows us to use some working parts of the infrastructure and manually provide what is left.

6.2.3 Typing systems with infinite contexts

Related with the choice of using the *Autosubst* library, we recall the use of infinite contexts, or contexts as functions mapping natural numbers to simple types. As already mentioned, this idea comes from the tutorial found in [25]. However, these are not the contexts that we work with in our paper proofs, thus, one would require a formal proof to admit this use. We did not invest much on this and much like the case of de Bruijn indices, we admit these facilities in order to invest our effort in the essential part of the metatheory. Using this definition for contexts makes our metatheory simpler and allows using the *Autosubst* operations and tactics already defined for substitutions (as the type for contexts coincides with the type for substitutions over simple types).

6.2.4 What is a subsystem?

In contrast to some decisions mentioned that facilitated our work of formalisation, the rigorous presentation of the canonical subsystem of $\boldsymbol{\lambda m}$ was one of our major efforts. The task of mechanising a subsystem motivated us in this direction. In [13, Chapter 3], system $\boldsymbol{\lambda\mathcal{P}}$ is introduced as an isolated system that uses expressions from another system ($\boldsymbol{\lambda\mathcal{P}h}$).

We instead started by separating two different systems. First we defined what a subsystem of $\boldsymbol{\lambda m}$ was: a subset of expressions with a closed notion of reduction and derivable sequent. Then, we wanted a different (yet isomorphic) representation of this same subsystem, with its appropriate notion of substitution. Defining a reduction over this "self-contained" subsystem and proving the isomorphism with the induced notion of reduction for the subsystem was the last ingredient for our complete description of the (untyped) canonical subsystem of $\boldsymbol{\lambda m}$ (for the typing systems we equally prove an isomorphism between the typing relations).

In terms of the mechanisation we found this useful, as we could work with a system with its own induction principles and many other dedicated definitions. The downside to this technique was the need to prove an isomorphism. Instead, we could have used the subset types from the *Rocq Prover*, but as we have already argued, this would become tiring because of the constant necessity for proving a designated predicate.

6.3 Future work

Lastly, we mention two directions that could be followed as a continuation of this dissertation.

A first direction would be to extend the metatheory considered in our exercise of formalisation. An initial approach could be to mechanise more metatheoretical results. Furthermore, we could extend our formalisation by enriching our systems with a more complex syntax (for example, system $\boldsymbol{\lambda m}$ is a subsystem of system $\boldsymbol{\lambda J^m}$ from [14]). A distinct and possible new way of work would involve generalising our typing systems beyond simple types - for instance, by incorporating polymorphism or dependent types.

A second and completely different direction would be to further explore the problems found in the mentioned libraries that aid the formalisation of syntax with binders. From our experience using the *Autosubst* library, we could suggest solutions or improvements that would provide more automation in our use cases.

Bibliography

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 31–46, 1989.

[2] A. Abel, G. Allais, A. Hameer, B. Pientka, A. Momigliano, S. Schäfer, and K. Stark. Poplmark reloaded: Mechanizing proofs by logical relations. *Journal of Functional Programming*, 29:e19, 2019. doi: 10.1017/S0956796819000170.

[3] A. Adams. *Tools and techniques for machine-assisted meta-theory*. PhD thesis, The University of St Andrews, 1997.

[4] B. Aydemir and S. Weirich. Lngen: Tool support for locally nameless representations. *Draft available at http://www. cis. upenn. edu/~ sweirich/papers/lngen*, 61:108, 2010.

[5] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: the POPLMark challenge. In *Theorem Proving in Higher Order Logics: 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005. Proceedings 18*, pages 50–65. Springer, 2005.

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

[7] H. Barendregt, W. Dekkers, and R. Statman. *Perspectives in logic: Lambda calculus with types*. Perspectives in logic. Cambridge University Press, Cambridge, England, June 2013.

[8] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[9] 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.

[10] J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order abstract syntax in coq. In *International Conference on Typed Lambda Calculi and Applications*, pages 124–138. Springer, 1995.

[11] A. Dudenhefner and D. Pautasso. Mechanized subject expansion in uniform intersection types for perpetual reductions. In *9th International Conference on Formal Structures for Computation and Deduction (FSCD 2024)*, pages 8–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.

[12] R. Dychkoff and L. Pinto. Cut-elimination and a permutation-free sequent calculus for intuitionistic logic. *Studia Logica*, 60:107–118, 1998.

[13] J. Espírito Santo. *Conservative extensions of the lambda-calculus for the computational interpretation of sequent calculus*. PhD thesis, University of Edinburgh, 2002.

[14] J. Espírito Santo and L. Pinto. A calculus of multiary sequent terms. *ACM Transactions on Computational Logic (TOCL)*, 12(3):1–41, 2011.

[15] Y. Forster, S. Schäfer, S. Spies, and K. Stark. Call-by-push-value in coq: operational, equational, and denotational theory. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 118–131, 2019.

[16] G. Gonthier. A computer-checked proof of the Four Color Theorem. Technical report, Inria, Mar. 2023. URL `https://inria.hal.science/hal-04034866`.

[17] H. Herbelin. A Lambda-calculus Structure Isomorphic to Gentzen-style Sequent Calculus Structure. In L. Pacholski and J. Tiuryn, editors, *Computer Science Logic, 8th International Workshop, CSL '94, Kazimierz, Poland, September 25-30, 1994, Selected Papers*, volume 933 of *Computer Science Logic, 8th International Workshop, CSL '94, Kazimierz, Poland, September 25-30, 1994, Selected Papers*, pages 61–75, Kazimierz, Poland, Sept. 1994. URL `https://inria.hal.science/inria-00381525`.

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

[19] Y. (https://stackoverflow.com/users/1809211/yves). Mutualy recursive function and termination checker in coq. Stack Overflow. URL `https://stackoverflow.com/questions/13286198/mutualy-recursive-function-and-termination-checker-in-coq/13288907#13288907`. Accessed: September, 2025.

[20] *Rocq Prover Reference Manual*. Inria and CNRS and contributors, 2025. URL `https://rocq-prover.org/doc/V9.0.0/refman/index.html`. Version 9.0.0.

[21] G. Lee, B. C. Oliveira, S. Cho, and K. Yi. Gmeta: A generic formal metatheory framework for first-order representations. In *European Symposium on Programming*, pages 436–455. Springer, 2012.

[22] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, Dec. 2009. doi: 10.1007/s10817-009-9155-4. URL `https://inria.hal.science/inria-00360768`.

[23] J. McKinna and R. Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3):373–409, 1999.

[24] F. Pottier. A coq library for dealing with binding using de bruijn indices. GitHub. URL `https://github.com/rocq-community/dblib`. Accessed: September, 2025.

[25] *Autosubst Manual*. Saarland University, 2016. URL https://www.ps.uni-saarland.de/autosubst/.

[26] S. Schäfer, T. Tebbi, and G. Smolka. Autosubst: Reasoning with de bruijn terms and parallel substitutions. In C. Urban and X. Zhang, editors, *Interactive Theorem Proving*, pages 359–374, Cham, 2015. Springer International Publishing. ISBN 978-3-319-22102-1.

[27] K. Stark, S. Schäfer, and J. Kaiser. Autosubst 2: reasoning with multi-sorted de bruijn terms and vector substitutions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 166–180, 2019.

[28] P. Urzyczyn, M. H. S. Rensen, and M. H. Sorensen. *Lectures on the Curry-Howard Isomorphism*. Studies in Logic and the Foundations of Mathematics. Elsevier Science & Technology, July 2006.

[29] R. Vestergaard and J. Brotherston. The mechanisation of barendregt-style equational proofs (the residual perspective). *Electronic Notes in Theoretical Computer Science*, 58(1):18–36, 2001.