

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**

Contents

1	Introduction	2
1.1	Motivation	2
1.2	Objectives	3
1.3	Document structure	4
2	Background	5
2.1	Simply typed λ -calculus	5
2.2	λ -calculus with de Bruijn syntax	8
2.3	Mechanising meta-theory in <i>Rocq</i>	10
3	Multiary λ-calculus and its canonical subsystem	18
3.1	The system λm	18
3.2	The canonical subsystem	20
3.3	Mechanisation in <i>Rocq</i>	24
4	Canonical λ-calculus	35
4.1	The system $\vec{\lambda}$	35
4.2	$\vec{\lambda}$ vs the canonical subsystem of λm	37
4.3	Conservativeness	42
4.4	Mechanisation in <i>Rocq</i>	45
5	The isomorphism $\lambda \cong \vec{\lambda}$	51
5.1	Mappings θ and ψ	51
5.2	Mechanisation in <i>Rocq</i>	55
6	Discussion	57

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 metatheory?” and “Why the multiary λ -calculus?”.

Before addressing these questions formally, we could just say that mechanising mathematics is an enjoyable task. And for what matters, this could be all we say about our motivation. All I intend to say is that even if our work in mathematics had no application or direct consequences, the fun of doing it would be a good enough motivation. Mechanising mathematics is like a computer game for a mathematician.

1.1.1 Why mechanise (metatheory)?

The mechanisation of metatheory has been taken seriously for at least 20 years [3]. 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 [11]. 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 [15].

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.

All of the latter are perfectly good motivations for the work done in this dissertation.

Then, the question of *Why metatheory?* arises. From the reasons above, some are highlighted by the task of mechanising metatheory. For example, it is often argued that metatheoretical proofs “*are long, contain few essential insights, and have a lot of tedious but error-prone cases*” [17].

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*).

1.1.2 Why the multiary λ -calculus?

In the beginning of [18, 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 [12], a multiary version of the λ -calculus (with explicit substitutions) is introduced, whose typing rules correspond to a fragment of the sequent calculus. Furthermore, the corresponding reduction rules for this system behave as cut elimination.

Considering a slightly different multiary version of the λ -calculus (and excluding explicit substitutions), one gets a system that was studied in detail in CMAT [9, 10], here named λm . The study of 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

The theoretical objectives for this dissertation are the study of:

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

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. More 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 objective related to the mechanisation is to implement every definition and proof as close as possible to the pen-and-paper versions, but also to present clean and simple implementations. Of course this may be challenging and conflictuous, but it is in general the objective behind any mechanisation of mathematics.

1.3 Document structure

This dissertation is structured in six chapters.

The second chapter serves as an introduction to the ordinary λ -calculus, also containing a second style of λ -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.

The third chapter introduces the system λm and its canonical subsystem, along with some simple results. It also includes a last chapter that provides a walkthrough of the mechanised definitions and proofs.

The fourth chapter independently introduces a new system called $\tilde{\lambda}$, that is isomorphic to the introduced canonical subsystem of λm . 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.

The fifth chapter is only about the isomorphism between the ordinary λ -calculus and system $\tilde{\lambda}$.

A last (sixth) chapter with the title of “Discussion” contains many directions that were not explored in the development of our work along with some further justification of the choices taken and some possible further work.

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 [4]. For what types and the simply typed lambda calculus is about we refer to [5] and [13].

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 , 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 $\lambda x y z.M$ instead of $\lambda x.(\lambda y.(\lambda z.M))$.
3. Multiple applications can be abbreviated as $M N_1 N_2$ instead of $(M N_1) N_2$.

Definition 2 (Free/bound 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 concepts take the term $M = x(\lambda x.x)$. The variable x occurs both free and bound in this term.

Definition 3 (α -equality). *We say that two λ -terms are α -equal when they are equal up to the names of variables in binders.*

Remark. *The previous informal definition lets us take advantage of a variable naming convention introduced below. With this notion of α -equality, the definition of substitution over λ -terms and meta-discussion of our syntax will be simplified. After defining the substitution operation we will rigorously introduce the definition for α -equivalence.*

Convention. *We will use the variable convention introduced in [4]. Without loss of generality, every λ -term that we refer from now on is chosen (via α -equality) to have different names for bound and free occurrences of variables.*

Definition 4 (Substitution). *For every λ -term M , we recursively define the substitution of N for the free occurrences of x in M , $M[x := N]$, as follows:*

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

Remark. *It is important to notice that by variable convention, the substitution operation described is capture-avoiding: the free variables in N will not be affected by the binders in M , as they are chosen to have different names.*

Definition 5 (Compatible Relation). *Let R be a binary relation on λ -terms. We say that R is compatible if it satisfies:*

$$\frac{(M_1, M_2) \in R}{(\lambda x.M_1, \lambda x.M_2) \in R} \quad \frac{(M_1, M_2) \in R}{(NM_1, NM_2) \in R} \quad \frac{(M_1, M_2) \in R}{(M_1 N, M_2 N) \in R}$$

Notation. *Given a binary relation R on λ -terms, we define:*

$$\begin{aligned} \rightarrow_R &\text{ as the compatible closure of } R; \\ \twoheadrightarrow_R &\text{ as the reflexive and transitive closure of } \rightarrow_R; \\ =_R &\text{ as the equivalence relation generated by } \twoheadrightarrow_R. \end{aligned}$$

Definition 6 (α -equivalence). Consider the following binary relation on λ -terms:

$$\alpha = \{(\lambda x.M, \lambda y.M[x := y]) \mid \text{for every } \lambda\text{-term } M \text{ and variable } y \text{ not occurring in } M\}.$$

We call α -equivalence to the equivalence relation $=_\alpha$.

Definition 7 (β -reduction). Consider the following binary relation on λ -terms:

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

We call one step β -reduction to the relation \rightarrow_β and multistep β -reduction to the relation \rightarrow_β^* .

Definition 8 (β -normal form). We say that a λ -term t is in β -normal form (or irreducible by \rightarrow_β) when there exists no λ -term t' such that

$$t \rightarrow_\beta t'.$$

Definition 9. We inductively define the sets of λ -terms NF and NA as follows:

$$\frac{}{x \in NA} \quad \frac{M_1 \in NA \quad M_2 \in NF}{M_1 M_2 \in NA} \quad \frac{M \in NA}{M \in NF} \quad \frac{M \in NF}{\lambda x.M \in NF}$$

Claim 1. Given a λ -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 $\boxed{(i) \Rightarrow (ii)}$ in the last section of this chapter. The proof for $\boxed{(ii) \Rightarrow (i)}$ is also mechanised in the script repository of our development.

2.1.2 Types

Definition 10 (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 isomorphism.

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 11 (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 12 (Context). A context Γ, Δ, \dots 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:

$$\begin{aligned} x : A, \dots, y : B & \text{ for } \{x : A, \dots, y : B\} \\ x : A, \dots, y : B, \Gamma & \text{ for } \{x : A, \dots, y : B\} \cup \Gamma. \end{aligned}$$

Definition 13 (Sequent). A sequent $\Gamma \vdash M : A$ is a triple of a context, a λ -term and a simple type.

Definition 14 (Typing rules for λ -terms). The following typing rules inductively define the notion of derivable sequent.

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

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 assistant and proposed a simplified syntax to deal with generic binders [7]. 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 15 (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.

$$\begin{aligned}\lambda x.x &\rightsquigarrow \lambda.0 \\ \lambda x.\lambda y.x &\rightsquigarrow \lambda.\lambda.0 \\ \lambda x.\lambda y.x &\rightsquigarrow \lambda.\lambda.1\end{aligned}$$

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

Definition 16 (Substitution). *A substitution σ, τ, \dots over nameless λ -terms is a function mapping natural numbers (indices) to nameless λ -terms.*

Here are some examples of useful substitutions.

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

Definition 17 (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:*

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

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 [17] and is very close to the actual mechanisation done using the *Autosubst* library.

Another variation we may encounter when formalising λ -terms using a nameless syntax is the typing system. A similar approach to our modification of the typing system can be found in [2, Chapter 7]. We formulate the definition of context and derivable sequents in the nameless setting as follows.

Definition 18 (Nameless context). *A nameless context Γ, Δ, \dots is a finite (possibly empty) sequence of simple types.*

Notation.

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

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

Definition 19 (Typing rules for nameless λ -terms).

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

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 λ -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*) [14] 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 [3]. 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 :=
| 0
| S (n: nat).
```

For example, the number 0 is represented by the constructor 0 and number 2 is represented as S (S 0). 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
  :  $\forall P : \text{nat} \rightarrow \text{Prop},$ 
     $P\ 0 \rightarrow (\forall n : \text{nat}, P\ n \rightarrow P\ (S\ n)) \rightarrow \forall n : \text{nat}, P\ n$ 
```

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 :
 $\forall a\ b\ c, a+(b+c) = (a+b)+c.$ 
```

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 induction 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` (abbreviated 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 [3] 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 [17, 16] 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 ($i \mapsto (\text{Var } i)$).
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 λ -calculus, as we will need it in chapter 5. The mechanisation is very straightforward and follows closely the examples given in [16, 17].

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 λ -calculus. 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 -relation (recall Definition 7). This inductive definition mechanises the β relation altogether with the compatibility closure (\rightarrow_β).

```
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 9).

```

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 $t \in \text{NF}$ satisfy. This will be our standard approach 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 [6, 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 induction 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` :

```
( $\forall s$ , normal  $s \rightarrow \sim$ exists  $t$ , step  $s t$ )
 $\wedge$ 
( $\forall s$ , apps  $s \rightarrow \sim$ 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 λ -terms (recall Definition 14 and Definition 19).

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

```
| Ax (x: var) (A: type) :
   $\Gamma x = A \rightarrow$  sequent  $\Gamma$  (Var  $x$ ) A
| Intro (t: term) (A B: type) :
  sequent (A.: $\Gamma$ ) t B  $\rightarrow$  sequent  $\Gamma$  (Lam t) (Arr A B)
```

```
| Elim (s t: term) (A B: type) :
  sequent  $\Gamma$  s (Arr A B)  $\rightarrow$  sequent  $\Gamma$  t A  $\rightarrow$  sequent  $\Gamma$  (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 [16], we use infinite contexts (contexts as infinite sequences). That way we can mechanise contexts as functions $\text{var} \rightarrow \text{type}$ (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 $(A : \Gamma)$. This is the same function we encountered when defining the substitution operation for the β -contractum $s.[u.\text{ids}]$.

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 [16]), 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` : $\forall \Gamma \ t \ A, \text{sequent } \Gamma \ t \ A \rightarrow$
 $\forall \Delta \ \xi, \Gamma = (\xi \ggg \Delta) \rightarrow \text{sequent } \Delta \ t.[\text{ren } \xi] \ A$

Chapter 3

Multitary λ -calculus and its canonical subsystem

This chapter introduces the main system that was studied in this dissertation: the multitary λ -calculus (λm). We introduce this system as the system $\lambda P h$ studied in [9, Chapter 3]. This system can also be found as λ^m in [10, Section 3], as a subsystem of λJ^m .

We provide an alternative description for a subsystem of h -normal forms of λm (corresponding to the system λP found in [9, Chapter 3]). At the end of this chapter one can find a detailed overview of the mechanisation done in this dissertation of the multitary λ -calculus and subsystems.

3.1 The system λm

First, we introduce some standard definitions for our system, like the grammar for λm -terms, a typical append operation on lists and substitution operation.

Definition 20 (λm -terms). *The λm -terms are simultaneously defined with λm -lists by the following grammar:*

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

We will refer to the union of λm -terms and λm -lists as λm -expressions.

Definition 21 (Append). *The append of two λm -lists, $l + l'$, is defined recursively on l as follows:*

$$\begin{aligned} [] + l' &= l', \\ (u :: l) + l' &= u :: (l + l'). \end{aligned}$$

Definition 22 (Substitution for λm -terms). *The substitution of a variable x by a λm -term v is mutually defined by recursion over λm -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 23 (Compatible Relation). Let R and R' be two binary relations on λm -terms and λm -lists respectively. We say they are compatible when they satisfy:

$$\begin{array}{c} \frac{(t, t') \in R}{(\lambda x.t, \lambda x.t') \in R} \quad \frac{(t, t') \in R}{(t(u, l), t'(u, l)) \in R} \quad \frac{(u, u') \in R}{(t(u, l), t(u', l)) \in R} \quad \frac{(l, l') \in R'}{(t(u, l), t(u, l')) \in R} \\[10pt] \frac{(u, u') \in R}{(u :: l, u' :: l) \in R'} \quad \frac{(l, l') \in R'}{(u :: l, u :: l') \in R'} \end{array}$$

Definition 24 (Reduction rules for λm -terms).

$$\begin{array}{c} \frac{}{((\lambda x.t)(u, []), t[x := u]) \in \beta_1} \quad \frac{}{((\lambda x.t)(u, v :: l), t[x := u](v, l)) \in \beta_2} \\[10pt] \frac{}{(t(u, l)(u', l'), t(u, l + (u' :: l')) \in h} \end{array}$$

Remark. As the compatible closure induces two relations, one on terms and the other on lists, we will use the notation \rightarrow_R for both these relations as we can get out of the context which one is being referenced.

Definition 25. The relation β will denote the relation $\beta_1 \cup \beta_2$. The same applies for the relation βh that will denote the relation $\beta \cup h$. Therefore, we will have the induced relations \rightarrow_β and $\rightarrow_{\beta h}$ (and analogous multistep relations \rightarrow_β^* and $\rightarrow_{\beta h}^*$).

We turn now our attention to the typing system of system λm . Given that λm 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 λm -term and a simple type. A sequent on lists $\Gamma; A \vdash l : B$ is a quadruple of a context, a simple type, a λm -list and another simple type.

Definition 27 (Typing Rules for λm -terms).

$$\begin{array}{c} \frac{}{x : A, \Gamma \vdash x : A} \text{Var} \quad \frac{x : A, \Gamma \vdash t : B}{\Gamma \vdash \lambda x.t : A \supset B} \text{Abs} \\[10pt] \frac{\Gamma \vdash t : A \supset B \quad \Gamma \vdash u : A \quad \Gamma; B \vdash l : C}{\Gamma \vdash t(u, l) : C} \text{mApp} \\[10pt] \frac{}{\Gamma; A \vdash [] : A} \text{Nil} \quad \frac{\Gamma \vdash u : A \quad \Gamma; B \vdash l : C}{\Gamma; A \supset B \vdash u :: l : C} \text{Cons} \end{array}$$

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:*

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

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

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

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

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 λm -terms t and t' , the following holds:*

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

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

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

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

Other classical results from λ -calculus like confluency and strong normalisation could also be proved for system λm , but are not covered in this dissertation.

3.2 The canonical subsystem

The canonical subsystem is a system within λm 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 λm -term t is in h -normal form when there exists no λm -term t' such that*

$$t \rightarrow_h t'.$$

Definition 29 (Canonical terms). *We inductively define the subsets of λm -terms and λm -lists, respectively Can and $CanList$, as follows:*

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

$$\frac{}{\square \in CanList} \quad \frac{u \in Can \quad l \in CanList}{u :: l \in CanList}$$

λm -terms $t \in Can$ are also called *canonical terms*. Analogously, λm -lists $l \in CanList$ are called *canonical lists*. Canonical expressions will refer to the set $Can \cup CanList$.

Similarly to what was done in chapter 2, we leave a claim stating that the canonical terms are exactly the λm -terms in h -normal form.

Claim 3. *Given a λm -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 $app : Can \times Can \times CanList \rightarrow Can$ 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 $l \in CanList$,*

$$t(u, l) \rightarrow_h app(t, u, l) \quad (\text{in } \lambda m).$$

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 λm -terms to their h -normal form.

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

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

Proposition 1 (Map h performs \rightarrow_h). *For every λm -term t ,*

$$t \rightarrow_h h(t),$$

and also, for every λm -list l ,

$$l \rightarrow_h h(l).$$

Proof. The proof proceeds easily by simultaneous induction on the structure of λm -expressions.

As h is defined using *app*, Lemma 3 is crucial for the case where t is an application. \square

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 λm , we will see how to induce a reduction relation for these canonical expressions given a reduction relation on λm -expressions.

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

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

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_β in λm . 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 $\lambda\mathbf{m}$ -terms u, u' such that $h(u) = t$ and $h(u') = t'$ and $u \rightarrow_\beta u'$. Then,

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

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 $\lambda\mathbf{m}$, by equipping 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:*

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

Also, from the previous definition, we may ask similar questions to those asked above about β -reduction for canonical expressions. For example, given $t \in \text{Can}$, 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 $\lambda\mathbf{m}$. Then,

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

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 $\lambda\mathbf{m}$ does not exactly coincide with system $\lambda\mathbf{P}$ from [9, Chapter 3.1]. We define a subsystem of $\lambda\mathbf{m}$ 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 $\lambda\mathbf{m}$ in the sense we have described before and an isomorphic system with its own syntax, substitution, reduction and typing rules (this is the system $\tilde{\lambda}$ 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 λm also crucially relies on the *Autosubst* library, and essentially follows the style adopted for the mechanisation of the simply typed λ -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 λm performed in this dissertation. The inductive type for the syntax of λm -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 23) 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 approach 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  $\beta_1$ : relation term :=
| Step_Beta1 (t: {bind term}) (t' u: term) :
  t' = t.[u :: ids]  $\rightarrow$   $\beta_1$  (mApp (Lam t) u []) t'.
```

```
Inductive  $\beta_2$ : relation term :=
| Step_Beta2 (t: {bind term}) (t' u v: term) l :
  t' = t.[u :: ids]  $\rightarrow$   $\beta_2$  (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')  $\rightarrow$  H (mApp (mApp t u l) u' l') (mApp t u l'').
```

Definition step := comp (union _ (union _ β_1 β_2) H).

Definition step' := comp' (union _ (union _ β_1 β_2) 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 ( $\Gamma$ : var $\rightarrow$ type) : term  $\rightarrow$  type  $\rightarrow$  Prop :=
| varAxiom (x: var) (A: type) :
   $\Gamma$  x = A  $\rightarrow$  sequent  $\Gamma$  (Var x) A
| Right (t: term) (A B: type) :
  sequent (A ::  $\Gamma$ ) t B  $\rightarrow$  sequent  $\Gamma$  (Lam t) (Arr A B)
| HeadCut (t u: term) (l: list term) (A B C: type) :
  sequent  $\Gamma$  t (Arr A B)  $\rightarrow$  sequent  $\Gamma$  u A  $\rightarrow$  list_sequent  $\Gamma$  B l C  $\rightarrow$ 
  sequent  $\Gamma$  (mApp t u l) C
with list_sequent ( $\Gamma$ :var $\rightarrow$ type) : type  $\rightarrow$  (list term)  $\rightarrow$  type  $\rightarrow$  Prop :=
| nilAxiom (C: type) : list_sequent  $\Gamma$  C [] C
| Lft (u: term) (l: list term) (A B C:type) :
  sequent  $\Gamma$  u A  $\rightarrow$  list_sequent  $\Gamma$  B l C  $\rightarrow$ 
  list_sequent  $\Gamma$  (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 :
  ( $\forall$  t t', step t t'  $\rightarrow \forall \Gamma$  A, sequent  $\Gamma$  t A  $\rightarrow$  sequent  $\Gamma$  t' A)
 $\wedge$ 
  ( $\forall$  l l', step' l l'  $\rightarrow \forall \Gamma$  A B, list_sequent  $\Gamma$  A l B  $\rightarrow$ 
    list_sequent  $\Gamma$  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 [16].

```
Lemma type_renaming :
   $\forall \Gamma,$ 
```

```

(∀ 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 $\Gamma \vdash A$:

```

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

```

The used *Autosubst* approach takes this notion of well-typed substitutions or context morphisms (see [17, 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 definitions 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` :

```
( $\forall t, \text{is\_canonical } t \rightarrow h\ t = t$ )
 $\wedge$ 
( $\forall l, \text{is\_canonical\_list } l \rightarrow \text{map } h\ l = l$ ).
```

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'  $\rightarrow$  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'  $\rightarrow$  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 \rightarrow type`) :

```
term  $\rightarrow$  type  $\rightarrow$  Prop :=
| Seq_CanTerm t A : sequent  $\Gamma$  t A  $\rightarrow$  canonical_sequent  $\Gamma$  (h t) A.
```

Inductive `canonical_list_sequent` (Γ : `var \rightarrow type`) :

```
type  $\rightarrow$  list term  $\rightarrow$  type  $\rightarrow$  Prop :=
| Seq_CanList l A B : list_sequent  $\Gamma$  A l B  $\rightarrow$ 
  canonical_list_sequent  $\Gamma$  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 [17], *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 [17].

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 [6, 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 l. destruct t.
- now apply HVar.
- apply HLam. now apply rec.
- apply HmApp.
  + now apply rec.
  + now apply rec.
  + assert (∀ l, Q l). {
    fix rec' l. destruct l0.
    - apply HNil.
    - apply HCons.
      + now apply rec.
      + now apply rec'. }
  now apply H.

```

Qed.

Proposition sim_list_ind : $\forall l, Q l$.

Proof.

```

fix rec l. 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 compatibility 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 := $\forall x y z:A, R x y \rightarrow R y z \rightarrow R x z$.

...

```
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 Compatibility.

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 Compatibility.

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 λm 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 [6, 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 [9, 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 approach on the formalisation of the canonical subsystem of λm 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 [9, 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 λ -calculus ($\vec{\lambda}$).

We call it canonical because it is in fact isomorphic to the canonical subsystem of λm introduced in the previous chapter. Moreover, it is a self-contained representation of the canonical subsystem (one can notice the similarities in the definitions when comparing to system λm). We will show this isomorphism in the second section. In the third section we give proof for the theorem of conservativeness, stating that λm is a conservative extension of $\vec{\lambda}$.

Furthermore, we call it a canonical λ -calculus because this system is also isomorphic to the simply typed λ -calculus. This is formally argued in chapter 5.

4.1 The system $\vec{\lambda}$

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

$$\begin{aligned} t, u &::= \text{var}(x) \mid \lambda x.t \mid \text{app}_v(x, u, l) \mid \text{app}_\lambda(x.t, u, l) \\ l &::= [] \mid u :: l \end{aligned}$$

Remark. *In the $\vec{\lambda}$ -terms there exist two different binding constructors: $\lambda x.t$ and $\text{app}_\lambda(x.t, u, l)$. In both constructors, every occurrence of $\text{var}(x)$ in the term t is bound and not free. This is, system $\vec{\lambda}$ has a dedicated constructor for the multiary application $(\lambda x.t)(u, l)$ in system λm .*

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

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

where the list append, $l + l'$, is defined similarly as in λm .

Now follows a strange definition for the substitution operation, as we have to be careful when dealing with a substitution over a constructor app_v .

Definition 36 (Substitution for $\vec{\lambda}$ -terms). *The substitution over a $\vec{\lambda}$ -term is mutually defined with the substitution over a $\vec{\lambda}$ -list as follows:*

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

Definition 37 (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:*

$$\begin{array}{c}
\frac{(t, t') \in R}{(\lambda x.t, \lambda x.t') \in R} \quad \frac{(t, t') \in R}{(\text{app}_\lambda(x.t, u, l), \text{app}_\lambda(x.t', u, l)) \in R} \\
\\
\frac{(u, u') \in R}{(\text{app}_\lambda(x.t, u, l), \text{app}_\lambda(x.t, u', l)) \in R} \quad \frac{(l, l') \in R'}{(\text{app}_\lambda(x.t, u, l), \text{app}_\lambda(x.t, u, l')) \in R} \\
\\
\frac{(u, u') \in R}{(\text{app}_v(x, u, l), \text{app}_v(x, u', l)) \in R} \quad \frac{(l, l') \in R'}{(\text{app}_v(x, u, l), \text{app}_v(x, u, l')) \in R} \\
\\
\frac{(u, u') \in R}{(u :: l, u' :: l) \in R'} \quad \frac{(l, l') \in R'}{(u :: l, u :: l') \in R'}
\end{array}$$

Lemma 4 (Compatibility lemmas). *Let R and R' be two binary relations on $\vec{\lambda}$ -terms and $\vec{\lambda}$ -lists respectively. If R and R' are compatible, then they satisfy:*

$$\begin{array}{c}
\frac{(l_1, l'_1) \in R'}{(l_1 + l_2, l'_1 + l_2) \in R'} \quad \frac{(l_2, l'_2) \in R'}{(l_1 + l_2, l_1 + l'_2) \in R'} \\
\\
\frac{(t, t') \in R}{(t@(u, l), t@(u, l)) \in R} \quad \frac{(u, u') \in R}{(t@(u, l), t@(u', l)) \in R} \quad \frac{(l, l') \in R'}{(t@(u, l), t@(u, l')) \in R}
\end{array}$$

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. \square

Definition 38 (Reduction rules for $\vec{\lambda}$ -terms).

$$\begin{aligned} app_{\lambda}(x.t, u, []) &\rightarrow_{\beta_1} t[x := u] \\ app_{\lambda}(x.t, u, v :: l) &\rightarrow_{\beta_2} t[x := u]@(v, l) \end{aligned}$$

Definition 39 (β -normal forms). We inductively define the sets of $\vec{\lambda}$ -terms and $\vec{\lambda}$ -lists in β -normal form, respectively NT and NL , as follows:

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

Remark. One could simply describe the β -normal forms of $\vec{\lambda}$ as the terms and lists with no occurrences of the constructor app_{λ} . This corresponds to idea of cut-elimination from sequent calculus and is one of the motivations for working with such systems. This system also offers an advantage in comparison to the λ -calculus, where a description of β -normal forms is more elaborated.

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

- (i) $t \in NT$;
- (ii) t is irreducible by \rightarrow_{β} .

We leave a similar claim to Claim 1 that will not be proved here. However, it will be used in the next chapter to provide an alternative argument for the bijection between β -normal forms of λ -terms and $\vec{\lambda}$ -terms.

Sequents in system $\vec{\lambda}$ may be defined similarly as in system λm . Then, we directly introduce the typing rules of this system.

Definition 40 (Typing Rules for $\vec{\lambda}$ -terms).

$$\begin{aligned} &\frac{}{x : A, \Gamma \vdash var(x) : A} \text{Var} \quad \frac{x : A, \Gamma \vdash t : B}{\Gamma \vdash \lambda x.t : A \supset B} \text{Abs} \\ &\frac{\Gamma, x : A \supset B \vdash u : A \quad \Gamma, x : A \supset B; B \vdash l : C}{\Gamma, x : A \supset B \vdash app_v(x, u, l) : C} \text{VarApp} \\ &\frac{\Gamma, x : A \vdash t : B \quad \Gamma \vdash u : A \quad \Gamma; B \vdash l : C}{\Gamma \vdash app_{\lambda}(x.t, u, l) : C} \text{LamApp} \\ &\frac{}{\Gamma; A \vdash [] : A} \text{Nil} \quad \frac{\Gamma \vdash u : A \quad \Gamma; B \vdash l : C}{\Gamma; A \supset B \vdash u :: l : C} \text{Cons} \end{aligned}$$

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

In this section we prove an isomorphism between $\vec{\lambda}$ and the canonical subsystem in λm .

Definition 41. Consider the following maps i and p :

$$\begin{aligned}
 i : \vec{\lambda}\text{-terms} &\rightarrow Can \\
 var(x) &\mapsto x \\
 \lambda x.t &\mapsto \lambda x.i(t) \\
 app_v(x, u, l) &\mapsto x(i(u), i'(l)) \\
 app_\lambda(x.t, u, l) &\mapsto (\lambda x.i(t))(i(u), i'(l)),
 \end{aligned}$$

where i' is simply defined as $i'([]) \mapsto []$ and $i'(u :: l) = i(u) :: i'(l)$;

$$\begin{aligned}
 p : \lambda m\text{-terms} &\rightarrow \vec{\lambda}\text{-terms} \\
 x &\mapsto var(x) \\
 \lambda x.t &\mapsto \lambda x.p(t) \\
 t(u, l) &\mapsto p(t)@(p(u), p'(l)),
 \end{aligned}$$

where p' is simply defined as $p'([]) \mapsto []$ and $p'(u :: l) = p(u) :: p'(l)$.

The following diagram summarizes the defined maps.

$$\begin{array}{ccc}
 & \lambda m & \\
 p \swarrow & & \downarrow h \\
 \vec{\lambda} & \xrightarrow{i} & Can
 \end{array}$$

We begin by proving that the diagram shown is commutative.

Lemma 5. Given $\vec{\lambda}$ -terms t, u and $\vec{\lambda}$ -list l ,

$$i(t@(u, l)) = app(i(t), i(u), i'(l)).$$

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

Theorem 2.

$$i \circ p = h$$

$$i' \circ p' = h'$$

Proof. The proof proceeds easily by simultaneous induction on the structure of the λm -term, using Lemma 5 in the application case. □

4.2.1 Bijection at the level of terms

Corollary 1.

$$\begin{aligned} i \circ p|_{Can} &= id_{Can} \\ i' \circ p'|_{CanList} &= id_{CanList} \end{aligned}$$

Proof. Each inversion is obtained via rewriting with Proposition 2 and then using Theorem 2. \square

Theorem 3.

$$\begin{aligned} p \circ i &= id_{\vec{\lambda}\text{-terms}} \\ p' \circ i' &= id_{\vec{\lambda}\text{-terms}} \end{aligned}$$

Proof. The proof proceeds easily by simultaneous induction on the structure of the $\vec{\lambda}$ -term. \square

4.2.2 Isomorphism at the level of reduction

In our subsystem 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 6. For every $\vec{\lambda}$ -terms t, u ,

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

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

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

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

For the case where $t = app_v(x, u, l)$, we use Lemma 5 to rewrite the term $i(t[x := v]) = i(v @ (u, l))$ as $app(i(v), i(u), i'(l))$. \square

Lemma 7. For every λm -terms t, u ,

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

and also, for every λm -term u and λm -list l ,

$$p'(l[x := u]) = p'(l)[x := p(u)].$$

Proof. The proof proceeds easily by simultaneous induction on the structure of the λm -term t . □

The following technical lemma says that we can derive the compatibility rules from the system $\vec{\lambda}$ given the canonical closure of compatible relation on λm .

Lemma 8. *Let R and R' be two binary relations on λm -terms and λm -lists respectively.*

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

$$I = \{(t, t') \mid i(t) (\rightarrow_R)_c i(t'), \text{ for every } \vec{\lambda}\text{-terms } t, t'\}$$

$$I' = \{(l, l') \mid i'(l) (\rightarrow_{R'})_c i'(l'), \text{ for every } \vec{\lambda}\text{-lists } l, l'\}$$

Proof. We provide proof for 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 , $(t, t') \in I \implies i(t) (\rightarrow_R)_c i(t')$.

Then, from the definition of the canonical closure relation, we have that there exist λm -terms t_1 and t_2 such that $h(t_1) = i(t)$ and $h(t_2) = i(t')$ and $t_1 \rightarrow_R t_2$.

We have:

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

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

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

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

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

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' ,*

$$t \rightarrow_\beta t' \implies i(t) (\rightarrow_\beta)_c i(t')$$

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

$$l \rightarrow_\beta l' \implies i'(l) (\rightarrow_\beta)_c i'(l').$$

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

Lemma 6 deals with substitution preservation in the β -reduction cases.

Lemma 8 deals with all the compatibility cases. □

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

$$t (\rightarrow_{\beta})_c t' \implies p(t) \rightarrow_{\beta} p(t')$$

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

$$l (\rightarrow_{\beta})_c l' \implies p'(l) \rightarrow_{\beta} p'(l').$$

Proof. The proof proceeds by simultaneous induction on the step relation of canonical terms.

Lemma 4 may be useful for compatibility steps.

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

4.2.3 Isomorphism at the level of typed terms

Lemma 9 (Append admissibility). The following rule is admissible in $\vec{\lambda}$:

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

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

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

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

Proof. The proof proceeds easily by inspection of t , using Lemma 9 when t is an application. □

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

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash_c i(t) : A} \quad \frac{\Gamma; A \vdash l : B}{\Gamma; A \vdash_c i'(l) : B}.$$

Proof. The proof proceeds easily by simultaneous induction on the typing rules of $\vec{\lambda}$. □

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

$$\frac{\Gamma \vdash_c t : A}{\Gamma \vdash p(t) : A} \quad \frac{\Gamma; A \vdash_c l : B}{\Gamma; A \vdash p'(l) : B}.$$

Proof. From Proposition 2 we have that $h(t) = t$ and $h'(l) = l$.

Then, inverting Definition 33, we have (in λm):

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

Therefore, the proof proceeds easily by simultaneous induction on the typing rules of λm .

Lemma 10 is crucial for the application case. □

Our argument for the isomorphism between the canonical subsystem in λm 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 λm .

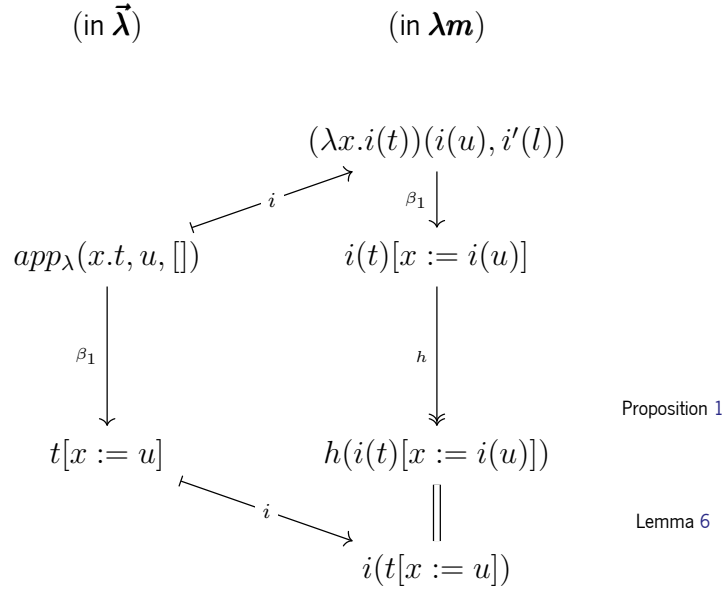
Theorem 8 (Conservativeness). *For every $\vec{\lambda}$ -terms t and t' , we have:*

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

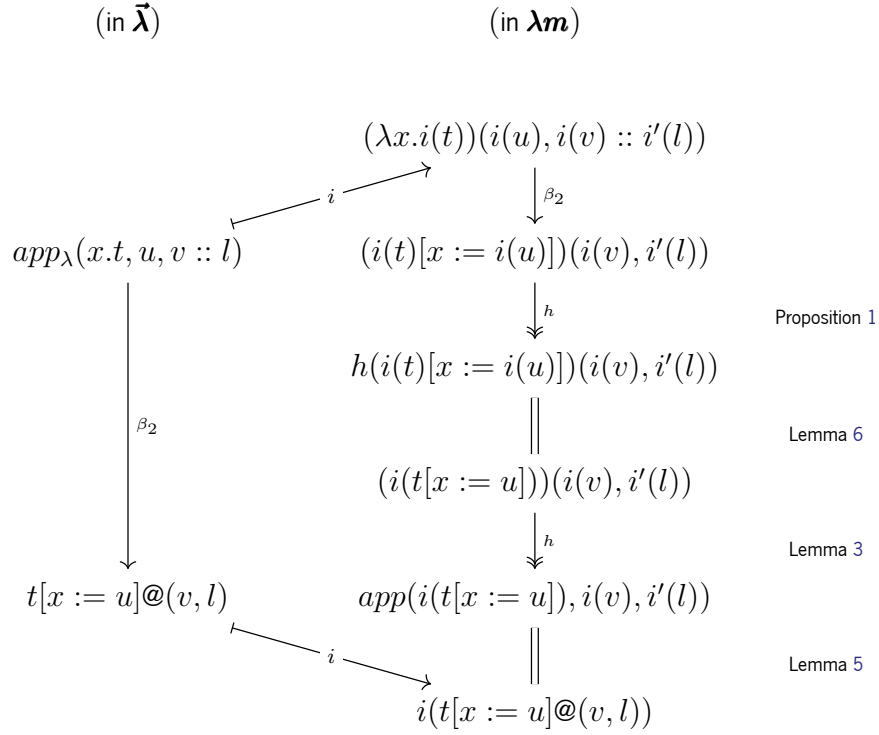
Proof. $\boxed{\implies}$ Let t and t' be $\vec{\lambda}$ -terms.

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

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



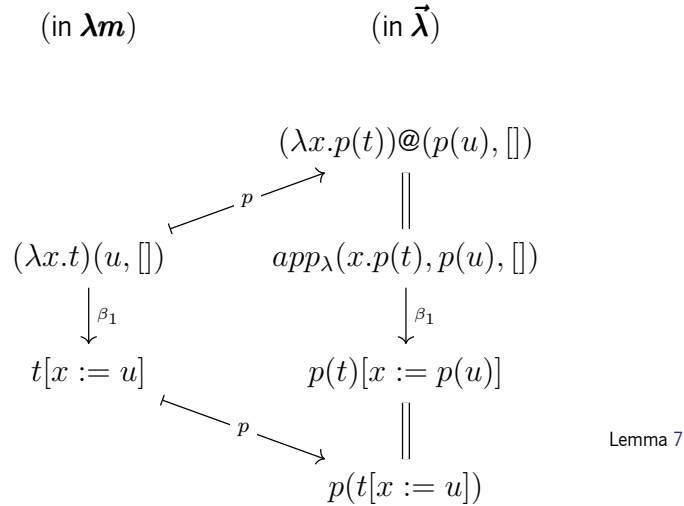
Case $t \rightarrow_{\beta_2} t'$:



$\boxed{\Leftarrow}$ Let t and t' be $\lambda\mathbf{m}$ -terms.

For this implication, we first show how a reduction $t \rightarrow_{\beta_h} t'$ in $\lambda\mathbf{m}$ is directly translated to a reduction $p(t) \rightarrow_{\beta} p(t')$ in $\vec{\lambda}$.

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



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

$$\begin{array}{ccc}
 (\text{in } \lambda\mathbf{m}) & & (\text{in } \vec{\lambda}) \\
 & & (\lambda x.p(t))@(p(u), p'(l)) \\
 & \swarrow p & \parallel \\
 (\lambda x.t)(u, v :: l) & & app_{\lambda}(x.p(t), p(u), p'(l)) \\
 \downarrow \beta_2 & & \downarrow \beta_2 \\
 t[x := u](v, l) & & p(t)[x := p(u)]@(p(v), p'(l)) \\
 & \swarrow p & \parallel \\
 & & p(t[x := u])@(p(v), p'(l))
 \end{array}
 \quad \text{Lemma 4 and Lemma 7}$$

Case $t \rightarrow_h t'$:

$$\begin{array}{ccc}
 (\text{in } \lambda\mathbf{m}) & & (\text{in } \vec{\lambda}) \\
 & & t(u, l)(u', l') \vdash p \rightarrow (p(t)@(p(u), p'(l)))@(p(u'), p'(l')) \\
 \downarrow h & & \parallel \\
 t(u, l + (u' :: l')) \vdash p \rightarrow p(t)@(p(u), p'(l + (u' :: l'))) & & \text{Simple induction on } p(t)
 \end{array}$$

From these cases, we proved that:

$$\begin{aligned}
 & t \twoheadrightarrow_{\beta_h} t' \implies p(t) \twoheadrightarrow_{\beta} p(t'), \text{ for every } \lambda\mathbf{m}\text{-terms } t, t' \\
 & (\text{which implies}) \quad i(t) \twoheadrightarrow_{\beta_h} i(t') \implies p(i(t)) \twoheadrightarrow_{\beta} p(i(t')), \text{ for every } \vec{\lambda}\text{-terms } t, t' \\
 & (\text{simplifying}) \quad i(t) \twoheadrightarrow_{\beta_h} i(t') \implies t \twoheadrightarrow_{\beta} t', \text{ for every } \vec{\lambda}\text{-terms } t, t'
 \end{aligned}$$

□

As a corollary of conservativeness, we can derive subject reduction for $\vec{\lambda}$ from $\lambda\mathbf{m}$.

Corollary 2 (Subject Reduction for $\vec{\lambda}$). *Given $\vec{\lambda}$ -terms t and t' , the following holds:*

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

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 proven admissible.

$$\begin{array}{c}
\text{Theorem 6} \frac{\Gamma \vdash t : A}{\Gamma \vdash_c i(t) : A} \\
\text{Inversion of Definition 33} \frac{\Gamma \vdash_c i(t) : A}{\Gamma \vdash t_0 : A} \quad t_0 \twoheadrightarrow_h h(t_0) \quad t \rightarrow_\beta t' \quad \text{Theorem 8} \\
\text{Theorem 1 with } \rightarrow \frac{\Gamma \vdash t_0 : A \quad t_0 \twoheadrightarrow_h h(t_0) \quad t \rightarrow_\beta t' \quad \text{Theorem 1 with } \rightarrow}{\Gamma \vdash i(t) : A \quad i(t) \twoheadrightarrow_{\beta h} i(t')} \\
\frac{\Gamma \vdash i(t') : A}{\Gamma \vdash_c h(i(t')) : A} \text{Definition 33} \\
\frac{\Gamma \vdash_c h(i(t')) : A}{\Gamma \vdash_c i(t') : A} \text{Proposition 2} \\
\frac{\Gamma \vdash_c i(t') : A}{\Gamma \vdash p(i(t')) : A} \text{Theorem 7} \\
\frac{\Gamma \vdash p(i(t')) : A}{\Gamma \vdash t' : A} \text{Theorem 3}
\end{array}$$

□

4.4 Mechanisation in Rocq

The mechanisations for the system $\vec{\lambda}$ follow the same style as the ones shown for the system λm using the *Autosubst* library, 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 exactly from the definitions for the system λm with the corresponding adaptations.

```

(* 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) →  $\beta_2$  (LambApp t u (v::l)) t'.
```

Definition step := comp (union _ β_1 β_2).

Definition step' := comp' (union _ β_1 β_2).

Definition multistep := clos_refl_trans_1n _ step.

Definition multistep' := clos_refl_trans_1n _ step'.

...

(* typing rules *)

Inductive sequent (Γ : var \rightarrow type) : term \rightarrow type \rightarrow Prop :=

```
| varAxiom (x: var) (A: type) :
```

```
   $\Gamma$  x = A  $\rightarrow$  sequent  $\Gamma$  (Vari x) A
```

```
| Right (t: term) (A B: type) :
```

```
  sequent (A ::  $\Gamma$ ) t B  $\rightarrow$  sequent  $\Gamma$  (Lamb t) (Arr A B)
```

```
| Left (x: var) (u: term) (l: list term) (A B C: type) :
```

```
   $\Gamma$  x = (Arr A B)  $\rightarrow$  sequent  $\Gamma$  u A  $\rightarrow$  list_sequent  $\Gamma$  B l C  $\rightarrow$ 
```

```
  sequent  $\Gamma$  (VariApp x u l) C
```

```
| KeyCut (t: {bind term}) (u: term) (l: list term) (A B C: type) :
```

```
  sequent (A ::  $\Gamma$ ) t B  $\rightarrow$  sequent  $\Gamma$  u A  $\rightarrow$  list_sequent  $\Gamma$  B l C  $\rightarrow$ 
```

```
  sequent  $\Gamma$  (LambApp t u l) C
```

with list_sequent (Γ :var \rightarrow type) : type \rightarrow (list term) \rightarrow type \rightarrow Prop :=

```
| nilAxiom (C: type) : list_sequent  $\Gamma$  C [] C
```

```
| Lft (u: term) (l: list term) (A B C:type) :
```

```
  sequent  $\Gamma$  u A  $\rightarrow$  list_sequent  $\Gamma$  B l C  $\rightarrow$ 
```

```
  list_sequent  $\Gamma$  (Arr A B) (u :: l) C.
```

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

This module also contains proofs for every compatibility lemma (recall Lemma 4).

Section CompatibilityLemmas.

Lemma step_comp_append1 :

```
 $\forall l1\ l1', \text{step}'\ l1\ l1' \rightarrow \forall l2, \text{step}'\ (l1 ++ l2)\ (l1' ++ l2).$ 
```

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 :
   $\forall v\ u\ u'\ l, \text{step } u\ u' \rightarrow \text{step } v@(u,l)\ v@(u',l).$ 
...
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 $\tilde{\lambda}$.

Let us see the statement of Lemma 8:

```

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 bureaucratic result, it helps simplifying further proofs and reveals some benefits of the approach taken, by formalising the concept of `is_compatible`, for example.

```

Theorem i_step_pres :
  ( $\forall (t\ t': \text{Canonical.term}),$ 
    Canonical.step t t'  $\rightarrow$  step_can (i t) (i t'))
   $\wedge$ 
  ( $\forall (l\ l': \text{list Canonical.term}),$ 
    Canonical.step' l l'  $\rightarrow$ 
    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.
...

```

As already mentioned in the last chapter, the mechanised version of Theorem 4 makes use of the automation provided by the `auto` tactic by strategically adding relevant lemmas to the proof context.

4.4.3 Conservativeness.v

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

Theorem `conservativeness` :

$\forall t \ t', \text{Canonical.multistep } t \ t' \leftrightarrow \text{LambdaM.multistep } (i \ t) \ (i \ t').$

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 $\tilde{\lambda}$ is the substitution operation. As referred upon Definition 36, we have a strange behaved substitution for the constructor app_v . In practice, on a substitution

$app_v(x, u, l)[x := v]$, there occurs an inspection of the term v that dictates the result of the substitution operation.

As we were working with a library that tries to automate definitions for substitution operations, we tested it in this case. But as expected, the `derive` tactic failed to give us the desired operation.

```
Subst_term =
(fix dummy ( $\sigma$  : var  $\rightarrow$  term) (s : term) {struct s} : term :=
match s as t return (annot term t) with
| Vari x  $\Rightarrow$  (fun x0: var  $\Rightarrow$   $\sigma$  x0) x
| Lamb t  $\Rightarrow$  (fun t0: {bind term}  $\Rightarrow$  Lamb t0.[up  $\sigma$ ]) t
| VariApp x u l  $\Rightarrow$  (fun (x0: var) (_: term) (_: list term)  $\Rightarrow$   $\sigma$  x0) x u l
| LambApp t u l  $\Rightarrow$ 
  (fun (t0: {bind term}) (s0: term) (l0: list term)  $\Rightarrow$ 
    LambApp t0.[up  $\sigma$ ] s0.[ $\sigma$ ] l0..[ $\sigma$ ]) 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  $\Rightarrow$  VariApp x u l
| Lamb t'  $\Rightarrow$  LambApp t' u l
| VariApp x u' l'  $\Rightarrow$  VariApp x u' (l' ++ u::l)
| LambApp t' u' l'  $\Rightarrow$  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  $\sigma$  s. change (annot term s). destruct s.
- exact ( $\sigma$  x).
- exact (Lamb (subst (up  $\sigma$ ) t)).
- exact (( $\sigma$  x)@(subst s, mmap (subst  $\sigma$ ) l)).
- exact (LambApp (subst (up  $\sigma$ ) t) (subst  $\sigma$  s) (mmap (subst  $\sigma$ ) l)).
```

Defined.

The downside to this was the need to manually prove every substitution lemma required by the *Auto-subst* instance `SubstLemmas`. This was crucial to enjoy the automation provided from the library for the mechanised inductive type of the $\vec{\lambda}$ -terms.

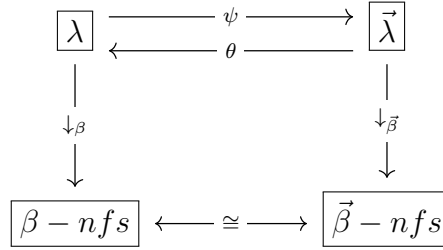
Chapter 5

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

In chapter 2, the simply typed λ -calculus was introduced.

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

This is of great interest as $\vec{\lambda}$ typing rules resemble a sequent calculus style. Thus, we have a correspondence between natural deduction (typing rules of λ -calculus) and a fragment of sequent calculus. The chapter is inspired in the works [8] and [9, Chapter 4]. The following diagram summarizes what we will be achieved in this chapter.



The $\vec{\beta}$ refers to the β -reduction steps in system $\vec{\lambda}$ and is used in the diagram to create a clear distinction.

5.1 Mappings θ and ψ

Definition 42. Consider the following maps θ and θ' :

$$\theta : \vec{\lambda}\text{-terms} \rightarrow \lambda\text{-terms}$$

$$\text{var}(x) \mapsto x$$

$$\lambda x.t \mapsto \lambda x.\theta(t)$$

$$\text{app}_v(x, u, l) \mapsto \theta'(x, u :: l)$$

$$\text{app}_\lambda(x.t, u, l) \mapsto \theta'(\lambda x.\theta(t), u :: l)$$

$$\theta' : (\lambda\text{-terms} \times \vec{\lambda}\text{-lists}) \rightarrow \lambda\text{-terms}$$

$$(M, []) \mapsto M$$

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

Definition 43. Consider the following map ψ' :

$$\begin{aligned}
 \psi' : (\lambda\text{-terms} \times \vec{\lambda}\text{-lists}) &\rightarrow \vec{\lambda}\text{-terms} \\
 (x, []) &\mapsto \text{var}(x) \\
 (x, u :: l) &\mapsto \text{app}_v(x, u, l) \\
 (\lambda x.M, []) &\mapsto \lambda x.\psi(M) \\
 (\lambda x.M, u :: l) &\mapsto \text{app}_\lambda(x.\psi(M), u, l) \\
 (MN, l) &\mapsto \psi'(M, \psi(N) :: l),
 \end{aligned}$$

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

5.1.1 Bijection at the level of terms

Lemma 11.

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

Proof. The proof proceeds by induction on the structure of λ -terms. □

Theorem 9.

$$\theta \circ \psi = \text{id}_{\lambda\text{-terms}}$$

Proof. The proof proceeds by induction on the structure of λ -terms and uses as lemma for the application case the ?? □

Theorem 10.

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

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

5.1.2 Isomorphism at the level of reduction

First, we need to introduce some lemmata that establish the preservation of substitution operations by the mappings θ , θ' and ψ' .

Lemma 12. For every $\vec{\lambda}$ -terms t, u and $\vec{\lambda}$ -list l ,

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

and also, for every λ -term M , $\vec{\lambda}$ -term u' and $\vec{\lambda}$ -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 $\vec{\lambda}$ -term t on the first proposition and on the structure of the $\vec{\lambda}$ -list l on the second proposition. \square

Corollary 3. For every λ -term M , $\vec{\lambda}$ -term u and $\vec{\lambda}$ -list l ,

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

Proof. The result follows as a corollary of Lemma 12, using Theorem 10 and Lemma 11 to rewrite the left-hand side of the equality. \square

Lemma 13. For every $\vec{\lambda}$ -terms t, u ,

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

and also, for every λ -term M , $\vec{\lambda}$ -term u and $\vec{\lambda}$ -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 $\vec{\lambda}$ -terms and -lists, using Lemma 12. \square

Lemma 14. For every λ -terms M, N and $\vec{\lambda}$ -list l ,

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

Proof. The proof follows by simultaneous induction on the structure of $\vec{\lambda}$ -terms and -lists, using corollary 3. \square

Now, we can state the isomorphism at the level of reduction.

Lemma 15. For every λ -terms M, N and $\vec{\lambda}$ -list l ,

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

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

Theorem 11. For every $\vec{\lambda}$ -terms t, t' ,

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

and also, for every λ -term M and $\vec{\lambda}$ -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 $\vec{\lambda}$ -terms.

Lemma 12 is useful for the cases of compatibility steps.

Lemma 13 is crucial for cases dealing with β steps. □

Theorem 12. For every λ -terms M, N and $\vec{\lambda}$ -list l ,

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

Proof. The proof proceeds by simultaneous induction on the structure of the step relation on λ -terms.

Lemma 14 is crucial for cases dealing with β steps. □

5.1.3 Isomorphism at the level of typed terms

Theorem 13 (θ admissibility). *The following rules are admissible:*

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

Proof. The proof proceeds easily by simultaneous induction on the structure of the typing rules of $\vec{\lambda}$ -terms. □

Theorem 14 (ψ' admissibility). *The following rule is admissible:*

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

Proof. The proof proceeds easily by induction on the structure of the typing rules of λ -terms. □

5.1.4 Bijection at the level of normal forms

For the following results recall both Definition 9 and Definition 39. We prove both bijections directly and give a hint on how they could be proved using the claims that we have provided. One can find similar results to this in [8].

Theorem 15.

$$\begin{aligned}\psi \circ \theta|_{NT} &= id_{NT} \\ \psi \circ \theta'|_{NA \times NL} &= \psi'|_{NA \times NL}\end{aligned}$$

Proof. The proof proceeds easily by simultaneous induction on the structure of $\vec{\lambda}$ -terms in NT and $\vec{\lambda}$ -lists in NL. □

Theorem 16.

$$\begin{aligned}\theta \circ \psi|_{NF} &= id_{NF} \\ \theta \circ \psi'|_{NA \times NL} &= \theta'|_{NA \times NL}\end{aligned}$$

Proof. The proof proceeds easily by simultaneous induction on the structure of λ -terms in NF and NA. □

Alternatively, we could prove that $M \in NF \implies \psi(M) \in NT$ and $t \in NT \implies \theta(t) \in NF$. Then, the shown theorems would automatically follow.

For example, from the assumption that $M \in NF$, using Claim 1, one gets that M is irreducible by \rightarrow_β . Then, from Theorem 12, $\psi(M)$ is also irreducible by \rightarrow_β (in $\vec{\lambda}$), which in turn means that $\psi(M) \in NT$ (by Claim 4).

5.2 Mechanisation in Rocq

In this section we provide a brief description of the mechanisations, as they follow from many previous definitions. Essentially, we just defined maps θ and ψ and mechanised every result.

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

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

```
Definition  $\theta'$  (s: Lambda.term) (l: list Canonical.term) :
  Lambda.term := fold_left (fun s v  $\Rightarrow$  App s ( $\theta$  v)) l s.
```


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

This was an undesired consequence of the use of polymorphic lists in the definition for $\vec{\lambda}$ -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. 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.

As the mechanised θ' is defined after θ , we have to consistently fold the definition for θ' in proofs to make them goal more readable. This can be seen in the mechanisation of Lemma 15:

Lemma $\theta'_{\text{step_pres}} \ 1 :$

$\forall s \ s', \text{Lambda.step } s \ s' \rightarrow \text{Lambda.step } (\theta' \ s \ 1) \ (\theta' \ s' \ 1).$

Proof.

```
induction l as [| u l]; intros ; asimpl ; try easy.
- fold ( $\theta'$  (App s ( $\theta$  u)) 1).
  fold ( $\theta'$  (App s' ( $\theta$  u)) 1).
  apply IHl. now constructor.
```

Qed.

Chapter 6

Discussion

Mechanisations in relation with the formalisations on the paper.

- Some ideas for the metatheory formalised come from attempts of mechanisations.
- The metatheory mechanised does not correspond exactly to the formalised in the literature.

An example: the polymorphic definition for system λm .

On the world of mechanising metatheory.

- Why use a outdated library for mechanising binders? What about *Autosubst 2*?
- Obstacles on the mechanisation of a non standard substitution operation.
- SSreflect style proofs for *Rocq Prover*.
- Could there be more automation?

Further work?

- Because of the modularity and of the *Autosubst* library we have the facility to enrich our typing systems (ex: SystemF?).

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. Adams. *Tools and techniques for machine-assisted meta-theory*. PhD thesis, The University of St Andrews, 1997.
- [3] 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.
- [4] H. Barendregt. *The lambda calculus*. Studies in Logic and the Foundations of Mathematics. Elsevier Science, London, England, 2 edition, Oct. 1987.
- [5] H. Barendregt, W. Dekkers, and R. Statman. *Perspectives in logic: Lambda calculus with types*. Perspectives in logic. Cambridge University Press, Cambridge, England, June 2013.
- [6] 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.
- [7] 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](https://doi.org/10.1016/1385-7258(72)90034-0).
- [8] R. Dychkoff and L. Pinto. Cut-elimination and a permutation-free sequent calculus for intuitionistic logic. *Studia Logica*, 60:107–118, 1998.
- [9] J. Espírito Santo. *Conservative extensions of the lambda-calculus for the computational interpretation of sequent calculus*. PhD thesis, University of Edinburgh, 2002.
- [10] J. Espírito Santo and L. Pinto. A calculus of multiary sequent terms. *ACM Transactions on Computational Logic (TOCL)*, 12(3):1–41, 2011.
- [11] G. Gonthier. A computer-checked proof of the Four Color Theorem. Technical report, Inria, Mar. 2023. URL <https://inria.hal.science/hal-04034866>.
- [12] 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>.
- [13] J. R. Hindley. *Basic Simple Type Theory*. Cambridge University Press, Cambridge, July 1997.
- [14] *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.
- [15] 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>.
- [16] *Autosubst Manual*. Saarland University, 2016. URL <https://www.ps.uni-saarland.de/autosubst/>.
- [17] 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.
- [18] P. Urzyczyn, M. H. S. Rensen, and M. H. Sørensen. *Lectures on the Curry-Howard Isomorphism*. Studies in Logic and the Foundations of Mathematics. Elsevier Science & Technology, July 2006.