Universidade do Minho
Escola de Ciências

Miguel José de Melo Alves

On the mechanisation of the multiary lambda calculus and subsystems

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

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

Setembro de 2025

Contents

Chapter 1

Introduction

1.1 Motivation

1.1.1 Why mechanise?

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 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 a really specific variation of the $\boldsymbol{\lambda}$-calculus could enable new ways of continuing the work being done in this topic.

1.1.2 Why the multiary λ-calculus?

In the begining of [18, Chapter 7.3], one can read *"Natural deduction proofs correspond to $\boldsymbol{\lambda}$-terms with types, and Hilbert style proofs correspond to combinators with types. What do sequent calculus proofs correspond to?"*. Many alternatives are given in this chapter, but none that can match the process of cut-elimination with normalisation.

In the novel paper of Herbelin [12], a multiary $\boldsymbol{\lambda}$-calculus (with explicit substitutions) is introduced, whose typing rules resemble a sequent calculus style of inference. Furthermore, the introduced reductions for this system correspond exactly with the process of cut elimination for a fragment of the sequent calculus.

Considering just the multiary version of the $\boldsymbol{\lambda}$-calculus (excluding explicit substitutions), one gets a system that was studied in detail in CMAT [10], often called $\boldsymbol{\lambda m}$. The study of the computational meaning behind the sequent calculus is one of the main motivations for considering systems such as $\boldsymbol{\lambda m}$, as they provide meaningful extensions for the ordinary $\boldsymbol{\lambda}$-calculus.

1.2 Objectives

The initial objectives of this dissertation were clear: to mechanise using the *Rocq Prover* a multiary version of the $\boldsymbol{\lambda}$-calculus and its canonical subsystem.

1.3 Document Structure

Chapter 2

Background

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

2.1 Simply typed λ-calculus

For the basic concepts and basic theory of the untyped $\boldsymbol{\lambda}$-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 ($\boldsymbol{\lambda}$-terms)**.** *The $\boldsymbol{\lambda}$-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 $\boldsymbol{\lambda}$-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 $\boldsymbol{\lambda}$-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 notation conventions on $\boldsymbol{\lambda}$-terms:*

1. *Outermost parentheses are omitted.*
2. *Multiple abstractions can be abbreviated as $\lambda xyz.M$ instead of $\lambda x.(\lambda y.(\lambda z.M))$.*
3. *Multiple applications can be abbreviated as MN_1N_2 instead of $(MN_1)N_2$.*

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

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

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

Definition 3 (α-equality)**.** *We say that two λ-terms are α-equal when they only differ in the name of their bound variables.*

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]. Every λ-term that we refer from now on is chosen (via α-equality) to have bound variables with different names from free variables.*

Definition 4 (Substitution)**.** *For every λ-term M, we recursively define the substitution of the free variable x by N 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. *Is is important to notice that by variable convention, the substitution operation described is capture-avoiding - bound variables will not be substituted ($x \in FV(M)$) and 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:*

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

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

$$\to_R \text{ as the compatible closure of } R;$$
$$\twoheadrightarrow_R \text{ as the reflexive and transitive closure of } \to_R;$$
$$=_R \text{ as the equivalence relation generated by } \twoheadrightarrow_R.$$

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

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

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

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

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

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

Definition 8. *We say that a* **λ***-term* t *is irreducible by* $\to_\beta$ *when there exists no* **λ***-term* t' *such that*

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

Definition 9 (β-normal forms)**.** *We inductively define the set of* **λ***-terms in* β*-normal form, NF, and normal applications, NA, as follows:*

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

Claim 1. *Given a* **λ***-term* M*, the following are equivalent:*

(i) $M \in$ *NF;*

(ii) M *is irreducible by* $\to_\beta$*.*

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

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 an atomic variable.*

Remark.

1. *A denumerable set of atomic variables is assumed and letters p, q, r range over this set.*
2. *It is important to notice that the symbol used for implication, $\supset$, is non standard in type theory. Rather it is used because of the literature in logic that we based our work on.*

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

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

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

$$x : A, \ldots, y : B \quad \textit{for} \quad \{x : A, \ldots, y : B\}$$
$$x : A, \ldots, y : B, \Gamma \quad \textit{for} \quad \{x : A, \ldots, y : B\} \cup \Gamma.$$

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 sequents.*

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

A sequent is derivable when it can be constructed by a successive application of the typing rules.

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 chapter will serve as a step closer to the mechanised version of the simply typed λ-calculus

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 $\boldsymbol{\lambda}$-terms)**.** *The nameless $\boldsymbol{\lambda}$-terms are defined by the following grammar:*

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

where i ranges over the natural numbers.

Remark. *Nameless $\boldsymbol{\lambda}$-terms have no α-conversion since there is no freedom to choose the names of bound variables.*

We may see some examples that illustrate the connection of ordinary and nameless syntax for $\boldsymbol{\lambda}$-terms.

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

Now, we will present a different formulation for the concept of substitution, adequate to deal with nameless $\boldsymbol{\lambda}$-terms.

Definition 16 (Substitution)**.** *A substitution over nameless $\boldsymbol{\lambda}$-terms is a function mapping natural numbers (indices) to nameless $\boldsymbol{\lambda}$-terms.*

Here are some examples of useful substitutions.

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

Definition 17 (Instantiation and composition)**.** *The operation of instantiating a substitution σ over a nameless $\boldsymbol{\lambda}$-term M, $M[\sigma]$, is recursively defined by the following equations:*

$$i[\sigma] = \sigma(i);$$
$$(\lambda.M)[\sigma] = \lambda.(M[0 \cdot (\uparrow \circ \sigma)]);$$
$$(M_1 M_2)[\sigma] = (M_1[\sigma])(M_2[\sigma]);$$

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

This definition for substitution instantiation 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 $\boldsymbol{\lambda}$-terms using a nameless syntax is the typing system. A similar aproach to our modification of the typing system can be found in [2, Chapter 7]. We reformulate the definition of context and derivable sequents as follows.

Definition 18 (Nameless context)**.** *A nameless context $\Gamma, \Delta, \ldots$ is a finite (possibly empty) sequence of simple types.*

Notation.

$|\Gamma|$ *is used to denote the size 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 $\boldsymbol{\lambda}$-terms)**.**

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

Remark. *For such strict definitions of contexts and typing rules we would require admissibility for structural rules (like the weakening rule shown below).*

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

That way, we show that our contexts as sequences may behave as multisets (as expected).

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 mechanisation of meta-theory. The multiary variations of the $\boldsymbol{\lambda}$-calculus that we are going to introduce will follow closely the basic approach described here with the corresponding adaptions.

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. We encode propositions as types and proofs for these propositions as programs in $\boldsymbol{\lambda}$-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 :=
| O
| S (n: nat).
```

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

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

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

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

b) Polymorphic lists

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

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

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

Here's a lemma on lists provided by the *Rocq* library:

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

This lemma relates two operations on lists:

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

Given their widespread utility, these operations will be often used in our mechanisations using lists.

2.3.2 Syntax with binders

A direct formalization 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's 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 won't 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 $\boldsymbol{\lambda}$-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) .
```

Here, the annotation `{bind term}` is an alias of the type `term`. We write this annotation to mark our constructors with binders in the syntax we want to mechanise.

This way, we may invoke the *Autosubst* classes, automatically deriving the desired instances.

```
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 function that maps every index into the corresponding variable term ($i \mapsto$ `(Var i)`).
2. Defining the recursive function that instantiates a variable renaming over a term.
3. Defining the recursive function that instantiates a parallel substitution over a term (using the already defined renamings).

Finally, there is also the proof for the substitution lemmas. Here, we see the power of this library, as the proofs for these lemmas are obtained automatically through the `derive` tactic.

2.3.4 Mechanising the simply typed λ-calculus

For this dissertation, we provide our own mechanisation of the simply typed $\boldsymbol{\lambda}$-calculus, as we will need it in chapter 5. The mechanisation is very straightforward and follows closely the examples given in [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 $\boldsymbol{\lambda}$-calculcus. The syntax for terms and *Autosubst* definitions were already presented and explained in the prior subsection.

The module then includes the definition for the one step β-relation (recall Definition 7). This inductive definition presents the β relation altogether with the compatibility closure.

```
Inductive step : relation term :=
| Step_Beta s s' t : s' = s.[t .: 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*. The type 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 defining 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 don't define directly a set NF of $\boldsymbol{\lambda}$-terms, but rather an inductive predicate that $\boldsymbol{\lambda}$-terms $t \in$ NF satisfy. This will be our standard aproach when mechanising subsets, because 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 indcution on the structure of a normal term, we need to have a simultaneous induction principle and prove two propositions simultaneously.

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

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

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

```
Theorem nfs_are_irreducible :
  (∀ s, normal s → ~exists t, step s t)
  ∧
  (∀ s, apps s → ~exists t, step s t).
Proof.
  apply mut_normal_ind ; intros.
```

```
  (* applying the combined induction principle *)
  - intro.
    apply H.
    destruct H0 as [t Ht].
    inversion Ht.
    now exists s'.
  - intro.
    apply H.
    destruct H0 as [t Ht].
    now exists t.
  - intro.
    now destruct H.
  - intro.
    destruct H1 as [t0 Ht0].
    inversion Ht0 ; subst.
    + inversion a.
    + apply H. now exists s'.
    + apply H0. now exists t'.
Qed.
```

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

The last thing our module contains is the typing rules for the $\boldsymbol{\lambda}$-terms (recall Definition 14 and Definition 19).

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

We directly mechanise the derivable sequent using an inductive definition (instead of defining sequents *a priori*).

Furthermore, using the approach in [16], we use infinite contexts (contexts as infinite sequences). That way we can mechanise contexts as functions `var→type` (the type of a parallel substitution over `type`) and take more advantage of the *Autosubst* definitions and tactics. Of course, in any derivation, only a

finite part of the (infinite) context is used.

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

As remarked upon the definition of the typing rules for the nameless terms, we still want to show admissibility for structural rules. We do this by proving the preservation of renamings (also an idea from [16]), as every structural rule involves a renaming of the nameless $\boldsymbol{\lambda}$-term (recall section 2.2).

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

Chapter 3

Multiary λ-calculus and its canonical subsystem

This chapter introduces the main system that was studied in this thesis: the multiary $\boldsymbol{\lambda}$-calculus ($\boldsymbol{\lambda m}$). We introduce this system as the system $\boldsymbol{\lambda \mathcal{P} h}$ found in [9, Chapter 3] and $\boldsymbol{\lambda^m}$ found in [10]. We also give an alternative description for a subsystem of h-normal forms in $\boldsymbol{\lambda m}$ (corresponding to the system $\boldsymbol{\lambda \mathcal{P}}$ found in [9, Chapter 3]). In the end of this chapter one can find a detailed overview of the mechanisations done.

3.1 The system λm

Definition 20 (Syntax of $\boldsymbol{\lambda m}$)**.** *The $\boldsymbol{\lambda m}$-terms and $\boldsymbol{\lambda m}$-lists are simultaneously defined by the following grammar:*

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

Definition 21 (Append)**.** *The append of two $\boldsymbol{\lambda m}$-lists, $l + l'$, is recursively defined as follows:*

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

Definition 22 (Substitution for $\boldsymbol{\lambda m}$-terms)**.** *The substitution over a $\boldsymbol{\lambda m}$-term is mutually defined with the substitution over a $\boldsymbol{\lambda m}$-list 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 $\boldsymbol{\lambda m}$-terms and $\boldsymbol{\lambda m}$-lists respectively. We say they are compatible when they satisfy:*

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

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

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

$$(\lambda x.t)(u, []) \to_{\beta_1} t[x := u]$$
$$(\lambda x.t)(u, v :: l) \to_{\beta_2} t[x := u](v, l)$$
$$t(u, l)(u', l') \to_h t(u, l + (u' :: l'))$$

By abuse of notation, we introduced the reduction rules with the notation of their compatible closure ($\to_R$).

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

Notation. *The relation β will denote the relation $\beta_1 \cup \beta_2$. The same for the relation βh that will denote the relation $\beta \cup h$. Therefore, we will have the induced relations $\to_\beta$ and $\to_{\beta h}$ (and analogous multistep relations $\twoheadrightarrow_\beta$ and $\twoheadrightarrow_{\beta h}$).*

The typing system of system $\boldsymbol{\lambda m}$ provides rules to derive two different kinds of sequents.

Definition 25 (Sequent)**.** *A sequent on terms $\Gamma \vdash t : A$ is a triple of a context, a $\boldsymbol{\lambda m}$-term and a simple type. A sequent on lists $\Gamma; A \vdash l : B$ is a quadruple of a context, a simple type, a $\boldsymbol{\lambda m}$-list and another simple type.*

Definition 26 (Typing Rules for $\boldsymbol{\lambda m}$-terms)**.**

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

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

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

Now we present classical results about the system $\boldsymbol{\lambda m}$, which culminate in the theorem of subject reduction.

Lemma 1. *The following rules are admissible:*

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

Proof. The proof proceeds by simultaneous induction on the structure of the typing rules. □

Lemma 2. *The following rule is admissible:*

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

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

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

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

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

Lemma 1 is used to prove the case $t \to_{\beta} t'$.

Lemma 2 is used to prove the case $t \to_{h} t'$. □

3.2 The canonical subsystem

We identify the set of $\boldsymbol{\lambda m}$-terms in h-normal form through the following inductive definition.

Definition 27 (Canonical terms)**.** *We inductively define the sets of $\boldsymbol{\lambda m}$-terms and $\boldsymbol{\lambda m}$-lists in h-normal form, respectively Can and $CanList$, as follows:*

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

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

$\boldsymbol{\lambda m}$-terms $t \in Can$ are also called canonical terms.

Now, we will describe how this subset of terms in $\boldsymbol{\lambda m}$ generates a subsystem.

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

Definition 28. *Given $t, u \in Can$ and $l \in CanList$, the operation $app(t,u,l)$ is defined by the following equations:*

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

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

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

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

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

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

Definition 29. *Consider the following map* h*:*

$$\begin{aligned} h : \boldsymbol{\lambda m}\textit{-terms} &\to Can \\ x &\mapsto x \\ \lambda x.t &\mapsto \lambda x.h(t) \\ t(u, l) &\mapsto app(h(t), h(u), h'(l)), \end{aligned}$$

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

Theorem 2. *For every* $\boldsymbol{\lambda m}$*-term* t*,*

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

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

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

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

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

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

Theorem 3 (h fixpoints)**.** *For every* $t \in Can$*,*

$$h(t) = t.$$

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

For the purpose of defining a subsystem of $\boldsymbol{\lambda m}$, we induce a reduction relation for these canonical terms given a reduction relation on the $\boldsymbol{\lambda m}$-terms and -lists.

Definition 30. *Let R and R' be two binary relations on $\boldsymbol{\lambda m}$-terms and $\boldsymbol{\lambda m}$-lists respectively. We inductively define the relations R_c and R'_c as follows:*

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

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

This definition allows us to define a β-reduction closed for the canonical terms, $(\to_\beta)_c$, derived from the relation $\to_\beta$ in $\boldsymbol{\lambda m}$. But this definition tells us nothing about the relation itself ...an interesting question is: how does this β-reduction behaves on canonical terms?

Given $t, u \in Can$, how do we reduce the canonical term $(\lambda x.t)(u,[])$ according to $(\to_\beta)_c$?

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

Given that $t, u \in Can$, we get that $(\lambda x.t)(u,[]) \in Can$. Therefore, from Theorem 3, we get that $(\lambda x.t)(u,[])\ (\to_\beta)_c\ h(t[x := u])$.

Furthermore, from this definition, we could even prove certain properties of $(\to_\beta)_c$.

For example:

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

This follows from inverting $t\ (\to_\beta)_c\ t'$. This is, there exist $\boldsymbol{\lambda m}$-terms u, u' such that $h(u) = t$ and $h(u') = t'$ and $u \to_\beta u'$.

$$\frac{\dfrac{u \to_\beta u'}{\lambda x.u \to_\beta \lambda x.u'}\ \text{(compatibility of } \to_\beta\text{)}}{h(\lambda x.u)\ (\to_\beta)_c\ h(\lambda x.u')}\ \text{(Definition 30)}$$

Simplifying h and rewriting $h(u)$ and $h(u')$ we conclude that $\lambda x.t\ (\to_\beta)_c\ \lambda x.t'$.

In the same spirit of Definition 30, we introduce the canonical typing closure.

Definition 31. *We inductively define the derivable sequents for canonical terms as follows:*

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

Also, from the previous definition, we may ask the same questions. For example, given $t \in Can$, is the following rule admissible?

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

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

Then,

$$\dfrac{\dfrac{x : A, \Gamma \vdash t' : B}{\Gamma \vdash \lambda x.t' : A \supset B}\ \text{Lam}}{\Gamma \vdash_c h(\lambda x.t') : A \supset B}\ \text{(Definition 31)}$$

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

We conclude our presentation of the canonical subsystem of $\boldsymbol{\lambda m}$. This presentation does not exaclty coincide with system $\boldsymbol{\lambda \mathcal{P}}$ from [9, Chapter 3.1].

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

3.3 Mechanisation in Rocq

The mechanisations for the system $\boldsymbol{\lambda m}$ follow almost the same style as the ones shown for the simply typed $\boldsymbol{\lambda}$-calculus in chapter 2 using the *Autosubst* library.

3.3.1 `LambdaM.v`

This module that contains the necessary definitions for the formalisations dealing with the system $\boldsymbol{\lambda m}$. The inductive type for the syntax of $\boldsymbol{\lambda m}$-terms is as follows.

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

The definition for $\boldsymbol{\lambda m}$-lists is hidden under the polymorphic list type `list term`. We give more details for this option at the end of this section.

To mechanise the reduction relations, we first defined the notion of compatibility as in Definition 23 and then the base step relations β_1, β_2 and h separately. That way we introduce the notions of compatible relation and also of compatible closure. This aproach is more elaborated than the one presented for the simply typed $\boldsymbol{\lambda}$-calculus and we also get into more details about these decisions at the end of this section.

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

Inductive β2: relation term :=
```

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

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

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

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

Here, the `comp` and `comp'` are the polymorphic relations on $\boldsymbol{\lambda m}$-terms and -lists respectively 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 have also the mechanised typing rules for $\boldsymbol{\lambda m}$, much in the style of what was done for the simply typed $\boldsymbol{\lambda}$-calculus.

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

3.3.2 `TypePreservation.v`

This module contains proof for Theorem 1 and necessary lemmas to prove it (recall Lemma 1).

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

Using *Autosubst*, we have to prove not only the preservation of types by the substitution operation but also by renamings. We prove these results using the techniques in the tutorial of [16].

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

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

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

The used *Autosubst* aproach takes this notion of well-typed substitutions or context morphisms (see [17, Chapter 4]) to generalise these lemmas.

As already mentioned, we use the combined induction principles for the proofs and need to declare the propositions using a conjunction on terms and lists.

3.3.3 IsCanonical.v

This module contains the necessary definitions for the formalisations dealing with the canonical subsystem of $\boldsymbol{\lambda m}$.

First, we define a predicate `is_canonical` that constructively defines the canonical terms in the style of Definition 27.

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

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

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


Fixpoint h (t: term) :=
  match t with
  | Var x      ⇒ Var x
  | Lam t      ⇒ Lam (h t)
```

```
  | mApp t u l ⇒ capp (h t) (h u) (map h l)
  end.
```

In the definition of map h we don't define map h', as we use the `map` function from the `List` library. The function `map h` behaves exactly as the intended map h'. Notice that this way we also avoid a mutually dependent definition.

In the *Rocq Prover*, we need to formally prove that the app operation and map h are closed for canonical terms. Of course that in description we have of the subsystem we easily argue this informally. For example, in the mechanised results, 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 every lemma and theorem presented in the description of the canonical subsystem. As an example, we show the mechanisation of Theorem 3.

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

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

The module ends with definitions for the reduction relation (recall Definition 30) and typing rules (recall Definition 31) for the canonical subsystem.

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

Definition step_can := canonical_relation step_beta.
Definition step_can' := canonical_list_relation step_beta'.
```

```
...
Inductive canonical_sequent (Γ: var→type) :
  term → type → Prop :=
| Seq_CanTerm t A : sequent Γ t A → canonical_sequent Γ (h t) A.
Inductive canonical_list_sequent (Γ: var→type) :
  type → list term → type → Prop :=
| Seq_CanList l A B : list_sequent Γ A l B →
                      canonical_list_sequent Γ A (map h l) B.
```

3.3.4 A closer look at the mechanisation

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

a) Mutually inductive types vs nested inductive types

Creating a mutually inductive type for the syntax of $\boldsymbol{\lambda m}$ 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.

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

  Proposition sim_term_ind : ∀t, P t.
  Proof.
    fix rec 1. destruct t.
    - now apply HVar.
    - apply HLam. now apply rec.
    - apply HmApp.
      + now apply rec.
      + now apply rec.
      + assert (∀l, Q l). {
            fix rec' 1. destruct l0.
            - apply HNil.
            - apply HCons.
              + now apply rec.
              + now apply rec'. }
        now apply H.
  Qed.

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

```
      + now apply rec.
  Qed.
End dedicated_induction_principle.
```

b) Formalising a compatible closure

Defining reductions in λ-calculus like systems always involve the notion of compatiblity closure, as we want to define reductions also in the subterms of a term.

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

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

We followed these definitions to define compatibility notions for the system $\boldsymbol{\lambda m}$ in a modular way. We define the compatible closure from a given base relation on $\boldsymbol{\lambda m}$-terms as follows:

```
Section Compatibilty.
  Variable base : relation term.

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

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

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

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

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

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

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

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

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

compatible.

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

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

c) Formalising a subsystem

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

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

Formalising the subset of terms using a predicate is the obvious way to do it. 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 mechanisations. 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)
...
```

Our aproach on the formalisation of such subsystem was to think of the canonical subsystem according to map h (defining reduction and typification using this map). After that, we may define a self-contained canonical system with its own syntax and definitions (in the spirit of [9, Chapter 3.1]). We do this with no reference to any definition from system $\boldsymbol{\lambda m}$ and prove that both representations are in fact isomorphic. That is the goal for chapter 4.

Chapter 4

Canonical λ-calculus

In this chapter we present a system that we give the name of canonical $\boldsymbol{\lambda}$-calculus ($\vec{\boldsymbol{\lambda}}$).

We call it canonical because it is in fact isomorphic to the canonical subsystem of $\boldsymbol{\lambda m}$ 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 $\boldsymbol{\lambda m}$). We will show this isomorphism in the second section. In the third section we give proof for the theorem of conservativeness, stating that $\boldsymbol{\lambda m}$ is a conservative extension of $\vec{\boldsymbol{\lambda}}$.

Furthermore, we call it a canonical $\boldsymbol{\lambda}$-calculus because this system is also isomorphic to the simply typed $\boldsymbol{\lambda}$-calculus. This is formally argued in chapter 5.

4.1 The system $\vec{\boldsymbol{\lambda}}$

Definition 32 (Syntax of $\vec{\boldsymbol{\lambda}}$)**.** *The $\vec{\boldsymbol{\lambda}}$-terms and $\vec{\boldsymbol{\lambda}}$-lists are simultaneously defined by the following grammar:*

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

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

Definition 33. *Given $\vec{\boldsymbol{\lambda}}$-terms t, u and $\vec{\boldsymbol{\lambda}}$-list l, the operation $t@(u, l)$ is defined by the following equations:*

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

where the list append, $l + l'$, is defined simlarly as in $\boldsymbol{\lambda m}$.

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 34 (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:*

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

Definition 35 (Compatible Relation)**.** *Let R and R' be two binary relations on $\vec{\lambda}$-terms and $\vec{\lambda}$-lists respectively. We say they are compatible when they satisfy:*

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

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

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

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

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:*

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

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

Proof. The proof proceeds easily by induction on lists for the append cases.

For the compatibility cases of @ operation, proof follows by inspection of the principle argument and application of the append cases. □

Definition 36 (Reduction rules for $\vec{\lambda}$-terms)**.**

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

Definition 37 (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:*

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

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 $\boldsymbol{\lambda}$-calculus, where a description of β-normal forms is more elaborated.*

Claim 2. *Given a $\vec{\lambda}$-term t, the following are equivalent:*

(i) $t \in NT$;

(ii) t is irreducible by $\to_\beta$.

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 $\boldsymbol{\lambda}$-terms and $\vec{\lambda}$-terms.

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

Definition 38 (Typing Rules for $\vec{\lambda}$-terms)**.**

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

$$\frac{\Gamma, x : A \supset B \vdash u : A \quad \Gamma, x : A \supset B; B \vdash l : C}{\Gamma, x : A \supset B \vdash app_v(x, u, l) : C}\ \textit{VarApp}$$

$$\frac{\Gamma, x : A \vdash t : B \quad \Gamma \vdash u : A \quad \Gamma; B \vdash l : C}{\Gamma \vdash app_\lambda(x.t, u, l) : C}\ \textit{LamApp}$$

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

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 $\boldsymbol{\lambda m}$.

Definition 39. *Consider the following maps* i *and* p*:*

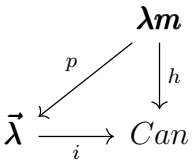
$$\begin{aligned} i : \vec{\boldsymbol{\lambda}}\text{-terms} &\to Can \\ var(x) &\mapsto x \\ \lambda x.t &\mapsto \lambda x.i(t) \\ app_v(x,u,l) &\mapsto x(i(u), i'(l)) \\ app_\lambda(x.t,u,l) &\mapsto (\lambda x.i(t))(i(u), i'(l)), \end{aligned}$$

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

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

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

The following diagram summarizes the defined maps.



We begin by proving that the diagram shown is commutative.

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

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

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

Theorem 4.

$$\begin{aligned} i \circ p &= h \\ i' \circ p' &= h' \end{aligned}$$

Proof. The proof proceeds easily by simultaneous induction on the structure of the $\boldsymbol{\lambda m}$-term, using Lemma 5 in the application case. □

4.2.1 Bijection at the level of terms

Corollary 1.

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

Proof. Each inversion is obtained via rewriting with Theorem 3 and then using Theorem 4. □

Theorem 5.

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

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

4.2.2 Isomorphism at the level of reduction

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

Lemma 6. *For every* $\vec{\boldsymbol{\lambda}}$*-terms* t, u*,*

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

and also, for every $\vec{\boldsymbol{\lambda}}$*-term* u *and* $\vec{\boldsymbol{\lambda}}$*-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 $\vec{\boldsymbol{\lambda}}$-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 $app(i(v), i(u), i'(l))$. □

Lemma 7. *For every* $\boldsymbol{\lambda m}$*-terms* t, u*,*

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

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

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

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

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

Lemma 8. *Let R and R' be two binary relations on $\boldsymbol{\lambda m}$-terms and $\boldsymbol{\lambda m}$-lists respectively.*

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

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

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, $(t, t') \in I \implies i(t)\ (\to_R)_c\ i(t')$.

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

We have:

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

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

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

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

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

Theorem 6. *For every $\vec{\boldsymbol{\lambda}}$-terms t, t',*

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

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

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

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 7. *For every* $t, t' \in Can$,

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

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

$$l\ (\to_\beta)_c\ l' \implies p'(l) \to_\beta p(l').$$

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* $\vec{\lambda}$*:*

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

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

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

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

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

Theorem 8 (i admissibility)**.** *For every* $\vec{\lambda}$*-term* t *and* $\vec{\lambda}$*-list* l*, the following rules are admissible:*

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

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

Theorem 9 (p admissibility)**.** *For every* $t \in Can$ *and* $l \in CanList$*, the following rules are admissible:*

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

Proof. From Theorem 3 we have that $h(t) = t$ and $h'(l) = l$.

Then, inverting Definition 31, we have (in $\boldsymbol{\lambda m}$):

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

Therefore, the proof proceeds easily by simultaneous induction on the typing rules of $\boldsymbol{\lambda m}$.

Lemma 10 is crucial for the application case. □

Our argument for the isomorphism between the canonical subsystem in $\boldsymbol{\lambda m}$ and $\vec{\boldsymbol{\lambda}}$ ends here.

From now on, we will use the self contained representation, system $\vec{\boldsymbol{\lambda}}$, to talk about canonical terms.

4.3 Conservativeness

The result of conservativeness establishes the connection between reduction in $\vec{\boldsymbol{\lambda}}$ and in $\boldsymbol{\lambda m}$.

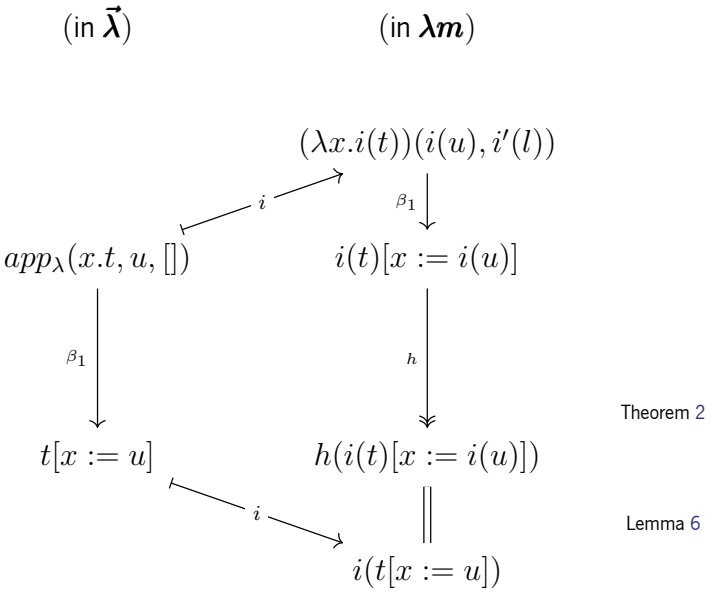
Theorem 10 (Conservativeness)**.** *For every $\vec{\boldsymbol{\lambda}}$-terms t and t', we have:*

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

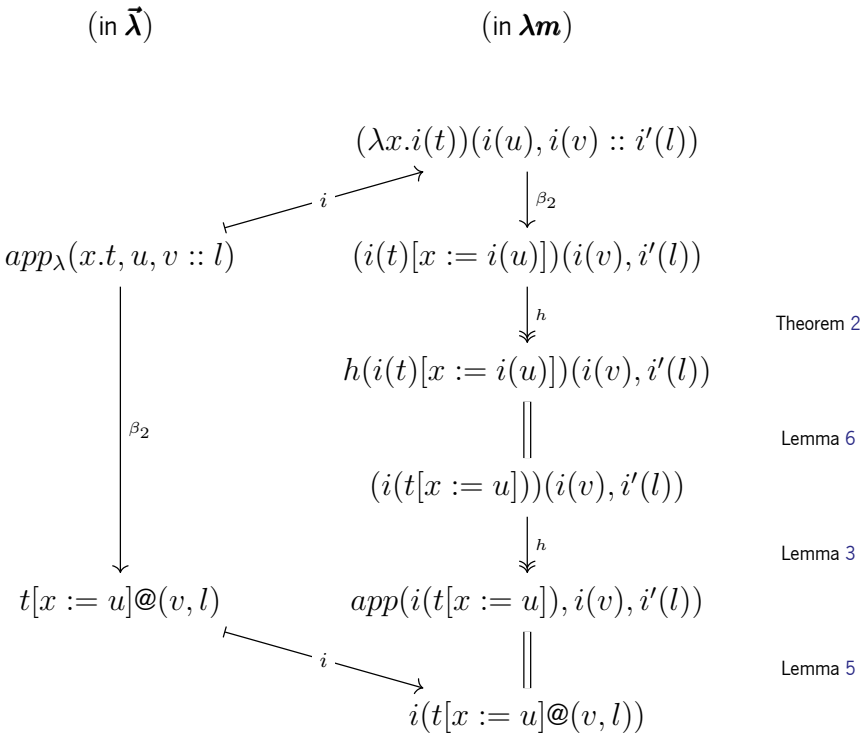
Proof. $\boxed{\Longrightarrow}$ Let t and t' be $\vec{\boldsymbol{\lambda}}$-terms.

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

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



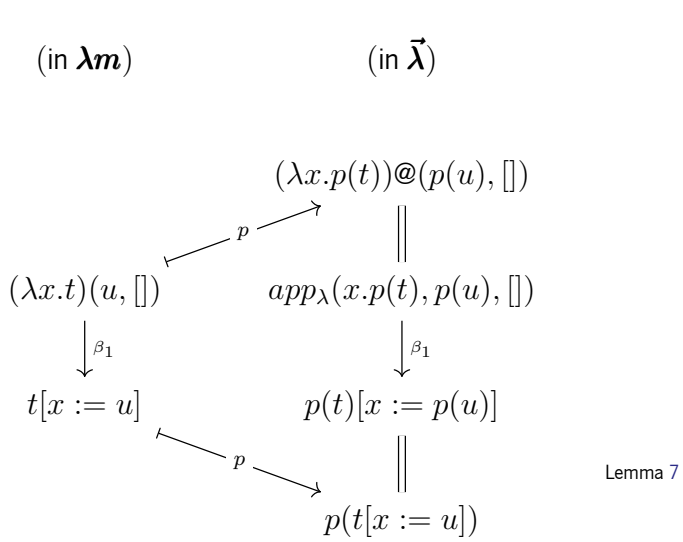
Case $t \to_{\beta_2} t'$:



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

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

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



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

(in $\boldsymbol{\lambda m}$) (in $\vec{\boldsymbol{\lambda}}$)

$$\begin{array}{ccc}
 & & (\lambda x.p(t))@(p(u), p'(l)) \\
 & \overset{p}{\swarrow\!\!\!\!\!\!\nearrow} & \| \\
(\lambda x.t)(u, v :: l) & & app_\lambda(x.p(t), p(u), p'(l)) \\
\downarrow{\scriptstyle \beta_2} & & \downarrow{\scriptstyle \beta_2} \\
t[x := u](v, l) & & p(t)[x := p(u)]@(p(v), p'(l)) \\
 & \overset{p}{\searrow\!\!\!\!\!\!\swarrow} & \| \qquad \text{Lemma 4 and Lemma 7} \\
 & & p(t[x := u])@(p(v), p'(l))
\end{array}$$

Case $t \to_h t'$:

(in $\boldsymbol{\lambda m}$) (in $\vec{\boldsymbol{\lambda}}$)

$$\begin{array}{ccc}
t(u, l)(u', l') & \overset{p}{\longmapsto} & (p(t)@(p(u), p'(l)))@(p(u'), p'(l')) \\
\downarrow{\scriptstyle h} & & \| \qquad \text{Simple induction on } p(t) \\
t(u, l + (u' :: l')) & \overset{p}{\longmapsto} & p(t)@(p(u), p'(l + (u' :: l')))
\end{array}$$

From these cases, we proved that:

$$\begin{array}{rl}
 & t \twoheadrightarrow_{\beta h} t' \implies p(t) \twoheadrightarrow_\beta p(t'), \text{ for every } \boldsymbol{\lambda m}\text{-terms } t, t' \\
\text{(which implies)} & i(t) \twoheadrightarrow_{\beta h} i(t') \implies p(i(t)) \twoheadrightarrow_\beta p(i(t')), \text{ for every } \vec{\boldsymbol{\lambda}}\text{-terms } t, t' \\
\text{(simplifying)} & i(t) \twoheadrightarrow_{\beta h} i(t') \implies t \twoheadrightarrow_\beta t', \text{ for every } \vec{\boldsymbol{\lambda}}\text{-terms } t, t'
\end{array}$$

□

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

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

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

Proof. The proof is shown in a derivation-like style. We use dashed lines for derivations that do not follow

from typing rules or rules proven admissible.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash t : A}{\Gamma \vdash_c i(t) : A}\ \text{Theorem 8}}{\Gamma \vdash t_0 : A}\ \text{Inversion of Definition 31} \quad t_0 \twoheadrightarrow_h h(t_0)}{\Gamma \vdash i(t) : A}\ \text{Theorem 1 with} \twoheadrightarrow \quad \dfrac{t \to_\beta t'}{i(t) \twoheadrightarrow_{\beta h} i(t')}\ \text{Theorem 10}}{\Gamma \vdash i(t') : A}\ \text{Theorem 1 with} \twoheadrightarrow}{\Gamma \vdash_c h(i(t')) : A}\ \text{Definition 31}}{\Gamma \vdash_c i(t') : A}\ \text{Theorem 3}}{\Gamma \vdash p(i(t')) : A}\ \text{Theorem 9}}{\Gamma \vdash t' : A}\ \text{Theorem 5}$$

□

4.4 Mechanisation in Rocq

The mechanisations for the system $\vec{\lambda}$ follow the same style as the ones shown for the system $\boldsymbol{\lambda m}$ 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 $\boldsymbol{\lambda m}$ 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) → β₂ (LambApp t u (v::l)) t'.

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

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

The mechanised step relations work as shown for the system $\boldsymbol{\lambda m}$ using a `comp` meta-relation for compatibility closure. In the next subsection we describe in more detail the aproach used to define the substitution operation for this system.

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

```
Section CompatibilityLemmas.
  Lemma step_comp_append1 :
    ∀l1 l1', step' l1 l1' → ∀l2, step' (l1 ++ l2) (l1' ++ l2).
  Proof.
    intros l1 l1' H.
```

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

4.4.2 `CanonicalIsomorphism.v`

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

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

```
Theorem i_step_pres :
  (∀(t t': Canonical.term),
      Canonical.step t t' → step_can (i t) (i t'))
  ∧
  (∀(l l': list Canonical.term),
      Canonical.step' l l' →
      step_can' (map i l) (map i l')).
Proof.
```

```
  pose step_can_is_compatible as Hic.
  destruct Hic.
  apply Canonical.mut_comp_ind ; intros ; auto.
  ...
```

As already mentioned in the last chapter, the mechanised version of Theorem 6 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 diagramatically in Theorem 10, divided into two parts, `conservativeness1` and `conservativeness2`, for each implication side.

```
Theorem conservativeness :
  ∀t t', Canonical.multistep t t' ↔ LambdaM.multistep (i t) (i t').
Proof.
  split.
  - intro H.
    induction H as [| t1 t2 t3].
    + constructor.
    + apply multistep_trans with (i t2) ; try easy.
      * now apply conservativeness1.
  - intro H.
    rewrite<- (proj1 inversion2) with t.
    rewrite<- (proj1 inversion2) with t'.
    induction H as [| t1 t2 t3].
    + constructor.
    + apply multistep_trans with (p t2) ; try easy.
      * now apply conservativeness2.
Qed.
```

4.4.4 A closer look at the mechanisation

a) Autosubst and a nonstandard substitution operation

One of the most peculiar definitions in system $\vec{\lambda}$ is the substitution operation. As referred upon Definition 34, 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 (σ : var → term) (s : term) {struct s} : term :=
match s as t return (annot term t) with
   | Vari x ⇒ (fun x0: var ⇒ σ x0) x
   | Lamb t ⇒ (fun t0: {bind term} ⇒ Lamb t0.[up σ]) t
   | VariApp x u l ⇒ (fun (x0: var) (_: term) (_: list term) ⇒ σ x0) x u l
   | LambApp t u l ⇒
       (fun (t0: {bind term}) (s0: term) (l0: list term) ⇒
        LambApp t0.[up σ] s0.[σ] l0..[σ]) t u l
   end)
```

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

```
Definition app (t u: term) (l: list term): term :=
  match t with
  | Vari x ⇒ VariApp x u l
  | Lamb t' ⇒ LambApp t' u l
  | VariApp x u' l' ⇒ VariApp x u' (l' ++ u::l)
  | LambApp t' u' l' ⇒ LambApp t' u' (l' ++ u::l)
  end.
Notation "t '@(' u ',' l ')'" := (app t u l) (at level 9).
...
Instance Ids_term : Ids term. derive. Defined.
Instance Rename_term : Rename term. derive. Defined.
Instance Subst_term : Subst term.
Proof.
  unfold Subst. fix inst 2. change _ with (Subst term) in inst.
  intros σ s. change (annot term s). destruct s.
  - exact (σ x).
  - exact (Lamb (subst (up σ) t)).
  - exact ((σ x)@(subst   s, mmap (subst σ) l)).
  - exact (LambApp (subst (up σ) t) (subst σ s) (mmap (subst σ) l)).
```

```
Defined.
```

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

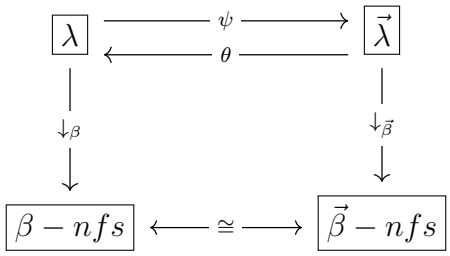
Chapter 5

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

In chapter 2, the simply typed $\boldsymbol{\lambda}$-calculus was introduced.

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

This is of great interest as $\vec{\boldsymbol{\lambda}}$ typing rules resemble a sequent calculus style. Thus, we have a correspondence between natural deduction (typing rules of $\boldsymbol{\lambda}$-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 $\vec{\boldsymbol{\lambda}}$ and is used in the diagram to create a clear distinction.

5.1 Mappings θ and ψ

Definition 40. *Consider the following maps θ and θ':*

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

$$\begin{aligned}
\theta' : (\boldsymbol{\lambda}\text{-terms} \times \vec{\boldsymbol{\lambda}}\text{-lists}) &\to \boldsymbol{\lambda}\text{-terms} \\
(M, []) &\mapsto M \\
(M, u :: l) &\mapsto \theta'(M\ \theta(u), l).
\end{aligned}$$

Definition 41. *Consider the following map ψ':*

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

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 $\boldsymbol{\lambda}$-terms. □

Theorem 11.

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

Proof. The proof proceeds by induction on the structure of $\boldsymbol{\lambda}$-terms and uses as lemma for the application case the **??**. □

Theorem 12.

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

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

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. □

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 12 and Lemma 11 to rewrite the left-hand side of the equality. □

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.

□

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.

□

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

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

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

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

Theorem 13. *For every $\vec{\boldsymbol{\lambda}}$-terms t, t',*

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

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

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

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

Lemma 12 is useful for the cases of compatibility steps.

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

Theorem 14. *For every $\boldsymbol{\lambda}$-terms M, N and $\vec{\boldsymbol{\lambda}}$-list l,*

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

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

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

5.1.3 Isomorphism at the level of typed terms

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

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

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

Theorem 16 (ψ' admissibility)**.** *The following rules is admissible:*

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

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

5.1.4 Bijection at the level of normal forms

For the following results recall both Definition 9 and Definition 37. 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 17.

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

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

Theorem 18.

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

Proof. The proof proceeds easily by simultaneous induction on the structure of $\boldsymbol{\lambda}$-terms in NF and NA. □

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

For example, from the assumption that $M \in \text{NF}$, using Claim 1, one gets that M is irreducible by $\to_\beta$. Then, from Theorem 14, $\psi(M)$ is also irreducible by $\to_\beta$ (in $\vec{\boldsymbol{\lambda}}$), which in turn means that $\psi(M) \in \text{NT}$ (by Claim 2).

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

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

The mechanised object that represents map θ' uses a higher order function on lists called `fold_left` that behaves exactly as θ', given the function `(fun s v ⇒ App s (θ v))` which folds the $\vec{\lambda}$-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 θ'_step_pres l :
  ∀ s s', Lambda.step s s' → Lambda.step (θ' s l) (θ' s' l).
Proof.
  induction l as [| u l]; intros ; asimpl ; try easy.
  - fold (θ' (App s (θ u)) l).
    fold (θ' (App s' (θ u)) l).
    apply IHl. now constructor.
Qed.
```

Chapter 6

Discussion

Mechanisations in relation with the formalisations on the paper.

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

  An example: the polymorphic definition for system $\boldsymbol{\lambda m}$.

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.

[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. Sorensen. *Lectures on the Curry-Howard Isomorphism*. Studies in Logic and the Foundations of Mathematics. Elsevier Science & Technology, July 2006.