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

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

1.2 Objectives

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

1.3 Document Structure

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

Chapter 2

Background

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

2.1 Simply typed λ-calculus

For the basic concepts and basic theory of the untyped **λ**-calculus we refer to [6]. For what types and the simply typed lambda calculus is about we refer to [5] and [11].

2.1.1 Syntax

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

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

where x denotes a variable.

Remark.

1. *A denumerable set of variables is assumed and letters x, y, z range over this set.*
2. *An abstraction is a **λ**-term of the kind $(\lambda x.M)$, that will bind occurrences of x in the term M, much like a function $x \mapsto M$.*
3. *An application is a **λ**-term of the kind $(M_1 M_2)$, where M_1 has the role of function and M_2 has the role of argument.*

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

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

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 [6]. 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 $\boldsymbol{\lambda}$-terms, we define:*

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

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

$=_R$ *as the equivalence relation generated by* $\twoheadrightarrow_R$.

Definition 6 (α-equivalence)**.** *Consider the following binary relation on $\boldsymbol{\lambda}$-terms:*

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

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

Definition 7 (β-reduction)**.** *Consider the following binary relation on $\boldsymbol{\lambda}$-terms:*

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

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 $\boldsymbol{\lambda}$-term t is irreducible by $\to_\beta$ when there exists no $\boldsymbol{\lambda}$-term t' such that $t \to_\beta t'$.*

Definition 9 (β-normal forms)**.** *We inductively define the set of $\boldsymbol{\lambda}$-terms in β-normal form, NF, and normal applications, NA, as follows:*

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

Claim 1. *Every $\boldsymbol{\lambda}$-term $t \in$ NF is irreducible by $\to_\beta$.*

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 type-assignement 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:*

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

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

Definition 14 (Typing rules for **λ**-terms)**.** *The following typing rules inductively define the notion of derivable 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 [8]. 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 **λ**-terms)**.** *The nameless **λ**-terms are defined by the following grammar:*

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

where i ranges over the natural numbers.

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

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

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

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

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

Here are some examples of useful substitutions.

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

Definition 17 (Instantiation and composition)**.** *The operation of instantiating a substitution σ over a nameless **λ**-term M, $M[\sigma]$, is recursively 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 [13] and are very close to the actual mechanisation done using the *Autosubst* library.

Another variation we may encounter when formalising λ-terms using a nameless syntax is the typing system. A similar aproach to our modification of the typing system can be found in [3, 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.*

Definition 19 (Typing rules for nameless λ-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 λ-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*) [12] 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 λ-calculus, in line with the Curry-Howard isomorphism.

It is arguably a great tool for mechanising meta-theory as it was widely used in the *POPLmark* challenge [4]. 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 0 → (∀n : nat, P n → P (S n)) → ∀n : nat, P n
```

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

```
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 : ∀l l', map (l++l') = (map l)++(map 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 [4] 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 [13, 1] 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 [2]. This theory comes into play for the implementation of the tactics `asimpl` and `autosubst` 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 for the chapter 5. The mechanisation is very straightforward and follows closely the examples given in [1, 13].

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').
```

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 λ-terms, but rather an inductive predicate that λ-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 [7, 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 [1], 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.

As a small display of the versatility of this options, in the `Intro` rule, one can find 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 [1]), as every structural rule involves a renaming of the nameless $\boldsymbol{\lambda}$-term.

```
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:*

$$t, u \ ::= \ x \mid \lambda x.t \mid t(u, l)$$
$$l ::= \ [] \mid u :: l.$$

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

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

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:*

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

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 $\twoheadrightarrow_{\beta h}$).*

Definition 25 (βh-normal forms)**.** *We inductively define the sets of $\boldsymbol{\lambda m}$-terms and $\boldsymbol{\lambda m}$-lists in βh-normal form, respectively NF and NL, as follows:*

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

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

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

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

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

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

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

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

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

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

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

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

As we have identified the βh-normal forms, we can also identify the set of h-normal forms, given by the following 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:*

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

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

We also call canonical terms to the $\boldsymbol{\lambda m}$*-terms in the set* Can*.*

Now, we will describe how this class 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. *For every* $\boldsymbol{\lambda m}$*-terms* t, u*, and* $\boldsymbol{\lambda m}$*-list* l*,*

$$t(u,l) \twoheadrightarrow_h app(t,u,l).$$

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:*

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

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. □

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

$$t = h(t).$$

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

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

This definition allows us to define a closed β reduction for the canonical terms, derived from the relation $\to_\beta$ in $\boldsymbol{\lambda m}$. This would result in the canonical relation $(\to_\beta)_c$.

In the same manner, we introduce the typing rules for canonical terms.

Definition 31 (Canonical typing rules)**.** *We inductively define the canonical typing rules, defined over every* $\boldsymbol{\lambda m}$*-term* t *and* $\boldsymbol{\lambda m}$*-list* l*:*

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

We conclude our presentation of the canonical subsystem of $\boldsymbol{\lambda m}$. This presentation does not exaclty coincide with [9]. We still want present a self-contained version of this subsystem, that we will call $\vec{\boldsymbol{\lambda}}$. We then prove that out self-contained version of the canonical terms is isomorphic to the susbsytem now described.

3.3 Mechanisation in Rocq

The mechanisations for the system $\boldsymbol{\lambda m}$ follow 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 on this in the end of this section.

Mechanising 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 in 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 λm-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 λm, much in the style of what was done for the simply typed λ-calculus.

```
Inductive sequent (Γ: var→type) : term → type → Prop :=
| varAξom (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 :=
| nilAξom (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 `IsCanonical.v`

This module contains the necessary definitions for the formalisations dealing with the canonical subsystem of λm.

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_is_surjective :
  (∀t, is_canonical t → t = h t)
  ∧
  (∀l, is_canonical_list l → l = map h 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.
```

3.3.3 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 λm in *Rocq* would be a simple task:

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

However, as reported in the final section of [13], *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 [13].

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 [7, 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 subsystem

A relevant part of the mechanisations done was to find simple representations for subsystems in the proof assitant.

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 [7, 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 proofs for every definition.

As an example, trying to define the substitution operation as in [9, Chapter 3.1] for the canonical terms mechanised using subset types, we would get:

```
(i hope i have this somewhere)
```

Our aproach for the formalisation of such subsystem was to think of the canonical subsystem using the map h that h-normalises $\boldsymbol{\lambda m}$-terms (defining reduction and typification using map h). After that, we could define a self-contained canonical system with its own syntax and definitions (in the spirit of [9, Chapter 3.1]) and prove both representations are isomorphic. That is the goal for the next chapter.

Chapter 4

Self-contained canonical system

4.1 The system $\vec{\lambda}$

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

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

Definition 33. *Given $\vec{\lambda}$-terms t, u and $\vec{\lambda}$-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 $\boldsymbol{\lambda m}$.

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 re-*

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

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

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

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

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

4.2 $\vec{\lambda}$ as a subsystem of λm

In this section we prove an isomorphism between $\vec{\boldsymbol{\lambda}}$ and the canonical terms in $\boldsymbol{\lambda m}$.

Definition 38. *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 maps defined.

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

We show some useful lemmas for the following results.

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. $\square$

4.2.1 Isomorphism at the level of terms

Theorem 4.

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

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

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

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

Proof. We provide proof for one of the compatibility cases:

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

From the definition of I, $(t, t') \in I \implies i(t) \to_{Rc} i(t')$.

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

We have:

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

Computing h, we get $(\lambda x.h(t_1))(h(i(u)), h'(i'(l))) \to_{Rc} (\lambda x.h(t_2))(h(i(u)), h'(i'(l)))$.

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

$$(\lambda x.h(t_1))(i(u), i'(l)) = (\lambda x.i(t))(i(u), i'(l)) = i(app_\lambda(x.t, u, l))$$
$$\to_{Rc} (\lambda x.h(t_2))(i(u), i'(l)) = (\lambda x.i(t'))(i(u), i'(l)) = i(app_\lambda(x.t', u, l))$$

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

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

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

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

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

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

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

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

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

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

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

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

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. $\boxed{\implies}$ 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 $t \to_{\beta_1} t'$:

$$
\begin{tikzcd}
\text{(in } \vec{\boldsymbol{\lambda}}\text{)} & \text{(in } \boldsymbol{\lambda m}\text{)} & \\
 & (\lambda x.i(t))(i(u), i'(l)) \arrow[d, "\beta_1"] & \\
app_\lambda(x.t, u, []) \arrow[ur, mapsto, "i"] \arrow[d, "\beta_1"] & i(t)[x := i(u)] \arrow[d, twoheadrightarrow, "h"] & \\
t[x := u] \arrow[dr, mapsto, "i"] & h(i(t)[x := i(u)]) \arrow[d, equal] & \text{Theorem 2} \\
 & i(t[x := u]) & \text{Lemma 6}
\end{tikzcd}
$$

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

$$
\begin{tikzcd}
\text{(in } \vec{\boldsymbol{\lambda}}\text{)} & \text{(in } \boldsymbol{\lambda m}\text{)} & \\
 & (\lambda x.i(t))(i(u), i(v) :: i'(l)) \arrow[d, "\beta_2"] & \\
app_\lambda(x.t, u, v :: l) \arrow[ur, mapsto, "i"] \arrow[ddd, "\beta_2"] & (i(t)[x := i(u)])(i(v), i'(l)) \arrow[d, twoheadrightarrow, "h"] & \text{Theorem 2} \\
 & h(i(t)[x := i(u)])(i(v), i'(l)) \arrow[d, equal] & \text{Lemma 6} \\
 & (i(t[x := u]))(i(v), i'(l)) \arrow[d, twoheadrightarrow, "h"] & \text{Lemma 3} \\
t[x := u]@(v, l) \arrow[dr, mapsto, "i"] & app(i(t[x := u]), i(v), i'(l)) \arrow[d, equal] & \text{Lemma 5} \\
 & i(t[x := u]@(v, l)) &
\end{tikzcd}
$$

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

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

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

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

$$
\begin{tikzcd}
 & (\lambda x.p(t))@(p(u),[]) \arrow[d, equal] \\
(\lambda x.t)(u,[]) \arrow[ur, mapsto, "p"] \arrow[d, "\beta_1"] & app_\lambda(x.p(t),p(u),[]) \arrow[d, "\beta_1"] \\
t[x := u] \arrow[dr, mapsto, "p"] & p(t)[x := p(u)] \arrow[d, equal] \\
 & p(t[x := u])
\end{tikzcd}
$$

Lemma 7

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

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

$$
\begin{tikzcd}
 & (\lambda x.p(t))@(p(u),p'(l)) \arrow[d, equal] \\
(\lambda x.t)(u,v::l) \arrow[ur, mapsto, "p"] \arrow[d, "\beta_2"] & app_\lambda(x.p(t),p(u),p'(l)) \arrow[d, "\beta_2"] \\
t[x := u](v,l) \arrow[dr, mapsto, "p"] & p(t)[x := p(u)]@(p(v),p'(l)) \arrow[d, equal] \\
 & p(t[x := u])@(p(v),p'(l))
\end{tikzcd}
$$

Lemma 4 and Lemma 7

Case $t \to_h t'$:

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

$$
\begin{tikzcd}
t(u,l)(u',l') \arrow[r, mapsto, "p"] \arrow[d, "h"] & (p(t)@(p(u),p'(l)))@(p(u'),p'(l')) \arrow[d, equal] \\
t(u,l+(u'::l')) \arrow[r, mapsto, "p"] & p(t)@(p(u),p'(l+(u'::l')))
\end{tikzcd}
$$

??

From these cases, we proved that:

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

□

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

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

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

Proof.

$$
\begin{prooftree}
\AxiomC{$\Gamma \vdash t : A$}
\LeftLabel{Theorem 8}
\UnaryInfC{$\Gamma \vdash_c i(t) : A$}
\dashedLine
\LeftLabel{Inversion of Definition 31}
\UnaryInfC{$\Gamma \vdash t_0 : A$}
\AxiomC{$t_0 \twoheadrightarrow_h h(t_0)$}
\dashedLine
\LeftLabel{Theorem 1 with $\twoheadrightarrow$}
\BinaryInfC{$\Gamma \vdash i(t) : A$}
\AxiomC{$t \to_{\beta} t'$}
\dashedLine
\RightLabel{Theorem 10}
\UnaryInfC{$i(t) \twoheadrightarrow_{\beta h} i(t')$}
\dashedLine
\RightLabel{Theorem 1 with $\twoheadrightarrow$}
\BinaryInfC{$\Gamma \vdash i(t') : A$}
\RightLabel{Definition 31}
\UnaryInfC{$\Gamma \vdash_c h(i(t')) : A$}
\dashedLine
\RightLabel{Theorem 3}
\UnaryInfC{$\Gamma \vdash_c i(t') : A$}
\RightLabel{Theorem 9}
\UnaryInfC{$\Gamma \vdash p(i(t')) : A$}
\dashedLine
\RightLabel{Theorem 5}
\UnaryInfC{$\Gamma \vdash t' : A$}
\end{prooftree}
$$

□

4.4 Mechanisation in Rocq

...

```
(* 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 :=
| varAξom (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 :=
| nilAξom (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.
```

4.5 A closer look at the mechanisation

...

Chapter 5

An isomorphism with the simply typed λ-calculus

$$\begin{tikzcd}
\boxed{\boldsymbol{\lambda}} \arrow[rr, leftrightarrow, "\cong"] \arrow[d, "\downarrow_\beta"] & & \boxed{\vec{\boldsymbol{\lambda}}} \arrow[d, "\downarrow_{\vec{\beta}}"] \\
\boxed{\beta - nfs} \arrow[rr, leftrightarrow, "\cong"] & & \boxed{\vec{\beta} - nfs}
\end{tikzcd}$$

In our background chapter, 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 and typing rules.

This is of great interest as $\vec{\boldsymbol{\lambda}}$ typing rules resemble a sequent calculus style. Thus, we have a correspondence of natural deduction (typing rules of $\boldsymbol{\lambda}$-calculus) and a fragment of sequent calculus.

5.1 Mappings θ and ψ

Definition 39. *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 40. *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 Isomorphism 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 Lemma 11. ☐

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'. Proofs of lemmas will now be omitted as they are all formalized in the proof assistant and usually proceed routinely.

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

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

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

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

The following lemma is obtained as a corollary.

Lemma 13. *For every $\boldsymbol{\lambda}$-term M, $\vec{\boldsymbol{\lambda}}$-term u and $\vec{\boldsymbol{\lambda}}$-list l,*

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

Lemma 14 states that θ preserves the substitution operation. We use Lemma 12 to prove this result.

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

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

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

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

Lemma 15 states that ψ preserves the substitution operation (taking $l = []$). We use Lemma 13 to prove this result.

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

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

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

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

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

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 if the step relation on $\vec{\boldsymbol{\lambda}}$-terms. Lemma 12 is useful for the cases of compatibility steps and Lemma 14 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 15 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:*

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

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. □

Chapter 6

Discussion

- distancia das provas em Rocq ao papel

  λm com substituicoes explicitas

- AUTOSUBST e overkill neste caso?

- variacoes na defn de substituicao?

- avoiding AUTOSUBST 2

- possiveis extensoes para tipos dependentes e polimorfismo usando AUTOSUBST? (mmap)

- theres a Coq world out there...

  SSreflect style? Bookeping e vários resultados sao estipulados nao exactamente como no papel

  automaçao

  andar para a frente e para trás com o código

- e preciso dar muitos nomes, chatice

- tentativa de ser consistente no estilo de definicoes e nomes, mas dificuldade

Bibliography

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

[2] 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.

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

[4] 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 p opl m ark 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.

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

[7] 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.

[8] 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.

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

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

[13] 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.