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

The following chapter introduces essential background to aid the reading of this dissertation. First, we introduce the well-known simply typed **λ**-calculus. Then, we delve into the theory on mechanisation of meta-theory, specifically in the context of our work. These concepts are introduced and motivated by the task of formalising the **λ**-calculus system introduced.

2.1 Simply typed λ-calculus

For the untyped lambda calculus descriptions we refer to [4]. For what types and the simply typed lambda calculus is about we refer to [3] and [8].

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 any variable, typically in the range of x, y, z.

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

1. *Outermost parenthesis are omitted.*
2. *Multiple abstractions can be abreviated as $\lambda xyz.M$ instead of $\lambda x.(\lambda y.(\lambda z.M))$.*
3. *Multiple applications can be abreviated 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. With this notion of* α*-equality, the definition of substitution over* $\boldsymbol{\lambda}$*-terms and meta-discussion of our syntax will be simplified. After defining the substitution operation we will rigorously introduce the definition for* α*-conversion.*

Convention. *We will use the variable convention introduced in [4]. Every* $\boldsymbol{\lambda}$*-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* $\boldsymbol{\lambda}$*-term* M*, we recursively define the substitution of the free variable* x *by* N *in* M*,* $M[x := N]$*, as follows:*

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

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* $\boldsymbol{\lambda}$*-terms. We say that* R *is compatible if it satisfies:*

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

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

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

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

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

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

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

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

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

Definition 8 (β-normal forms)**.** *We inductively define the set of* $\boldsymbol{\lambda}$*-terms in* β*-normal form, NF, and normal applications, NA, as follows:*

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

These $\boldsymbol{\lambda}$*-terms are irreducible according to* $\to_\beta$*.*

2.1.2 Types

Definition 9 (Simple Types)**.** *The simple types are defined by the following grammar:*

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

where p *denotes any atomic variable, typically in the range of* p, q, r*.*

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 10 (Context)**.** *A context,* $\Gamma, \Delta, \ldots$*, is a partial function from the variables of* $\boldsymbol{\lambda}$*-terms to simple types.*

Notation.

1. *We may often refer to the partial function of as the set of pairs* (x, A) *written as* $x : A$*.*
2. *We will also simplify the set notation of contexts as follows:*

$$\mapsto \{\}$$
$$x : A \mapsto \{x : A\}$$
$$x : A, \Gamma \mapsto \{x : A\} \cup \Gamma$$

Definition 11 (Typing Rules for $\boldsymbol{\lambda}$-terms)**.** *A type-assignment or sequent is a triple,* $\Gamma \vdash M : A$*, that is inductively defined by the following inference rules (or typing rules):*

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

2.2 Mechanising meta-theory in Rocq

Having introduced the ordinary λ-calculus, we take it as our goal of formalisation. This helps motivating the main decisions behind our mechanisations. The variations of λ-calculus that we are going to introduce will follow closely the approach described here with the corresponding adaptions.

2.2.1 The Rocq Prover

The *Rocq Prover* is an interactive theorem prover. This is a tool that helps in the formalisation of mathematical results and that can generate machine-verified proofs via interaction with a human.

...

2.2.2 Syntax with binders

Formalising the untyped λ-calculus syntax in *Rocq* would result in an inductive definition similar to:

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

The question that every similar definition imposes is the definition of 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 variable definition can get rather exhausting when it comes to rigorously define all this syntactical aspects and substitution operations.

There are several alternatives described in the literature of mechanisation of meta-theory. The POPLmark challenge [2] points to the topic of binding as central for discussing the potential of modern-day proof assistants. From the many alternatives, we chose to focus in the nameless syntax proposed by de Bruijn.

2.2.3 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 [6]. This approach is claimed to be good for meta-lingual discussion and for the computer and computer programme. In contrast, this syntax is further away from the human reader.

The main idea is to treat variables as indices (represented by natural numbers) and to interpret these indices as the distance to the respective binder. Therefore, we will call these terms nameless.

Definition 12 (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.*

Now, we will construct a different formulation for the concept of substitution.

Definition 13 (Substitution)**.** *A substitution over nameless **λ**-terms is a function mapping natural numbers (indices) to **λ**-terms.*

Here are some examples of useful substitutions:

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

Definition 14 (Subsitution instantiation)**.** *The operation of instantiating a substitution σ over a nameless **λ**-term M, $M[\sigma]$, is defined recursively by the following equations:*

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

2.2.4 Autosubst library

The *Autosubst* library for the *Rocq Prover* simplifies the formalisation of syntax with binders. It provides the *Rocq Prover* with tactics to define substitution over an inductively defined syntax. Furthermore, it even offers some automation for proofs dealing with substitution lemmas.
It is supported over three main ingredients:

1. nameless (de Bruijn) syntax ;

2. parallel substitutions ;
3. explicit substitutions for automation.

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 in order to mark our binders in the syntax we want to formalise.
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 of the substitution lemmas. Here, we see the power of this library: this process is done automatically, using the provided `derive` tactic.

2.2.5 Mechanising λ-calculus

We define the one step β-reduction altogether with the compatibility steps:

```
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').
```

Formalising the typing system:

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

The typing system definition is based on [1].

Chapter 3

Multiary λ-calculus and subsystems

3.1 The multiary λ-calculus (λm)

Definition 15 ($\boldsymbol{\lambda m}$-terms)**.** *The $\boldsymbol{\lambda m}$-terms are 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 16 (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 17 (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 18 (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 19 (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 20 (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:*

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

Definition 21 (Typing Rules for $\boldsymbol{\lambda m}$-terms)**.**

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

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

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

3.1.1 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 22 (h-normal forms)**.** *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 23. *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 1. *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 24. *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 1. *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 1 is crucial for the case where t is an application. □

Theorem 2 (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 25 (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:*

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

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

Definition 26 (Canonical Typing System)**.** *We inductively define the canonical type-assignement, 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 [7]. 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.

3.1.2 Subject reduction for λm

Lemma 2 (Substitution Admissibility)**.** *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 3 (Append Admissibility)**.** *The following rules 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 3 (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 $\to_{\beta h}$.
(i) We easily prove the case $t \to_{\beta} t'$ using substitution admissability in Lemma 2.
(ii) We easily prove the case $t \to_{h} t'$ using append admissability in Lemma 9. □

3.2 The system $\vec{\lambda}$

Definition 27 ($\vec{\lambda}$-terms)**.** *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 28. *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 29 (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 30 (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 31 (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 32 (Typing Rules for $\vec{\lambda}$-terms)**.**

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

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

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

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

3.3 $\vec{\lambda}$ as a subsystem of λm

Definition 33. *Consider the following maps i and p:*

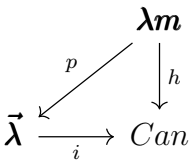
$$\begin{aligned} i : \vec{\boldsymbol{\lambda}}\text{-terms} &\to Can \\ var(x) &\mapsto x \\ \lambda x.t &\mapsto \lambda x.i(t) \\ app_v(x,u,l) &\mapsto x(i(u), i'(l)) \\ app_\lambda(x.t,u,l) &\mapsto (\lambda x.i(t))(i(u), i'(l)), \end{aligned}$$

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.



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. □

3.3.1 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 2 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. □

3.3.2 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:

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

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 $\vec{\lambda}$-terms t, t',*

$$t \to_\beta t' \implies i(t) \to_{\beta_c} i(t')$$

and also, for every $\vec{\lambda}$-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. □

3.3.3 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 $\vec{\boldsymbol{\lambda}}$. □

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

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

Proof. From Theorem 2 we have that $t = h(t)$ and $l = h'(l)$.
Then, inverting Definition 26, we have (in $\boldsymbol{\lambda m}$):

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

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.

3.4 Conservativeness

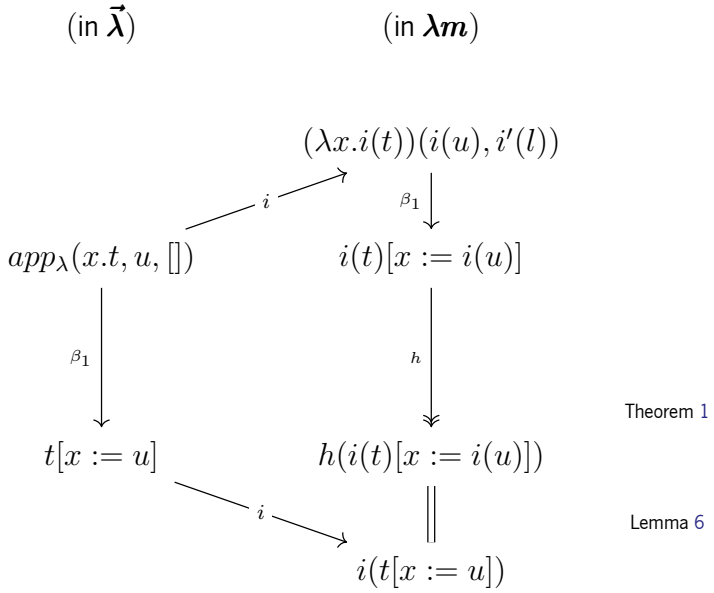
The result of conservativeness establishes the connection between reduction in $\vec{\boldsymbol{\lambda}}$ and in $\boldsymbol{\lambda m}$.

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

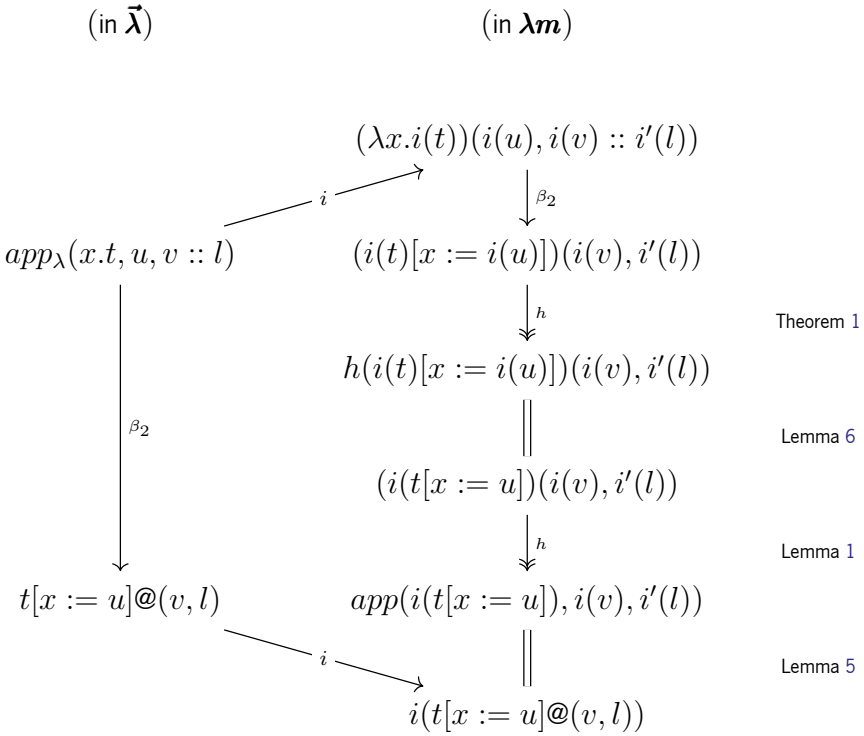
$$t \twoheadrightarrow_\beta t' \iff i(t) \twoheadrightarrow_{\beta h} i(t')$$

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

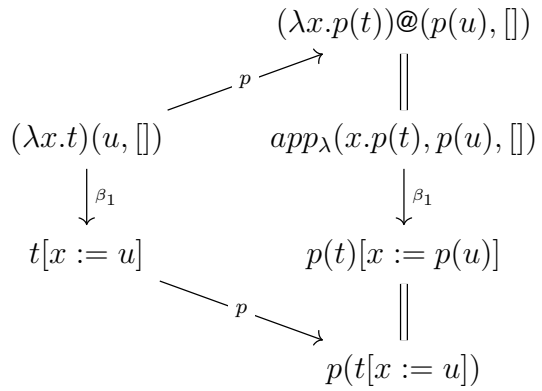


$\boxed{\Longleftarrow}$ Let t and t' be $\boldsymbol{\lambda m}$-terms.

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 $\vec{\lambda}$.

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

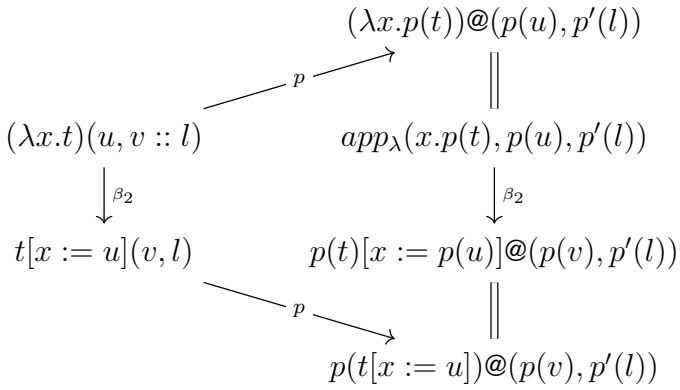
(in $\boldsymbol{\lambda m}$) (in $\vec{\boldsymbol{\lambda}}$)



Lemma 7

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

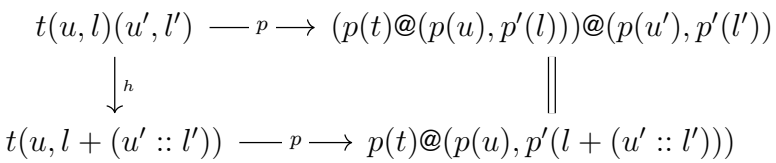
(in $\boldsymbol{\lambda m}$) (in $\vec{\boldsymbol{\lambda}}$)



Lemma 4 and Lemma 7

Case $t \to_h t'$:

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



??

From these cases, we proved that:

$$t \twoheadrightarrow_{\beta h} t' \implies p(t) \twoheadrightarrow_{\beta} p(t'), \text{ for every } \boldsymbol{\lambda m}\text{-terms } t, t'$$

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

$$\text{(simplifying)} \quad i(t) \twoheadrightarrow_{\beta h} i(t') \implies t \twoheadrightarrow_{\beta} t', \text{ for every } \vec{\boldsymbol{\lambda}}\text{-terms } t, t'$$

□

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.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash t : A}{\Gamma \vdash_c i(t) : A}\ \text{Theorem 8}}{\Gamma \vdash t_0 : A}\ \text{Inversion of Definition 26} \qquad t_0 \twoheadrightarrow_h h(t_0)}{\Gamma \vdash i(t) : A}\ \text{Theorem 3 with } \twoheadrightarrow \qquad \dfrac{t \to_{\beta} t'}{i(t) \twoheadrightarrow_{\beta h} i(t')}\ \text{Theorem 10}}{\Gamma \vdash i(t') : A}\ \text{Theorem 3 with } \twoheadrightarrow}{\Gamma \vdash_c h(i(t')) : A}\ \text{Definition 26}}{\Gamma \vdash_c i(t') : A}\ \text{Theorem 2}}{\Gamma \vdash p(i(t')) : A}\ \text{Theorem 9}}{\Gamma \vdash t' : A}\ \text{Theorem 5}$$

□

3.5 Mechanisation in Rocq

...

```
(* syntax *)
Inductive term: Type :=
| Var (x: var)
| Lam (t: {bind term})
| mApp (t: 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 (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'.
...
(* typing rules *)
Inductive sequent (gamma: var→type) : term → type → Prop :=
| varAxiom (x: var) (A: type) :
  gamma x = A → sequent gamma (Var x) A
| Right (t: term) (A B: type) :
  sequent (A .: gamma) t B → sequent gamma (Lam t) (Arr A B)
| HeadCut (t u: term) (l: list term) (A B C: type) :
  sequent gamma t (Arr A B) → sequent gamma u A → list_sequent gamma B l C →
  sequent gamma (mApp t u l) C

with list_sequent (gamma:var→type) : type → (list term) → type → Prop :=
| nilAxiom (C: type) : list_sequent gamma C [] C
| Lft (u: term) (l: list term) (A B C:type) :
  sequent gamma u A → list_sequent gamma B l C →
  list_sequent gamma (Arr A B) (u :: l) C.
```

3.6 A closer look at the mechanisation

In this section, we discuss several differences between the formalisations on the proof assistant and those presented on the literature. As we have already discussed binding and de Bruijn notation, we are not taking this into account from now on.

3.6.1 Mutually inductive types vs Nested inductive types

Creating a mutually inductive definition for $\boldsymbol{\lambda m}$ in *Rocq* is 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 [9], 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 custom 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 [9].
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 [5]. 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.
```

...

3.6.2 Formalising a subsystem

A relevant part of the mechanisation, was to represent subsystems in the proof assistant in a simple way. We isolate a subsyntax of $\boldsymbol{\lambda m}$ by defining a predicate over its terms:

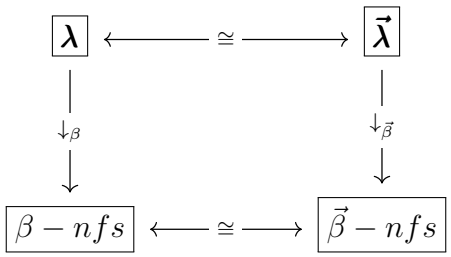
```
Inductive is_canonical: term → Prop :=
| cVar (x: var) : is_canonical (Var x)
| cLam (t: {bind term}) : is_canonical t → is_canonical (Lam t)
| cVarApp (x: var) (u: term) (l: list term) :
  is_canonical u → is_canonical_list l → is_canonical (mApp (Var x) u l)
| cLamApp (t: {bind term}) (u: term)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

This is the same idea that we introduced when inductively defining the sets of $\boldsymbol{\lambda m}$-terms Can and of $\boldsymbol{\lambda m}$-lists $CanList$.

a) Digression over subset types

Chapter 4

An isomorphism with the simply typed λ-calculus



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.

4.1 Mappings θ and ψ

Definition 34. *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 35. *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 $\psi'(M, [])$.

4.1.1 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. □

4.1.2 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 $\vec{\lambda}$-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',

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

The following lemma is obtained as a corollary.

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

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

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

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

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

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

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

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

Theorem 13. *For every $\vec{\lambda}$-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. □

4.1.3 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. □

Chapter 5

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] 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.

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

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

[5] 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.

[6] 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.

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

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

[9] 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.