Acknowledgments

Thanks your peoples here.

Statement of Integrity

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Resumo

Abstract em português.

Palavras-chave 3 a 5 palavras-chave, ordenadas alfabeticamente e separadas por vírgulas

Abstract

Your abstract here.

Keywords 3-5 keywords alphabetically ordered and comma-separated.

"We adore chaos because we love to produce order."

M. C. Escher

Contents

Chapter 1

Introduction

1.1 Motivation

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 $\boldsymbol{\lambda}$-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 $\boldsymbol{\lambda}$-calculus system introduced.

2.1 Lambda Calculus

2.1.1 Syntax

[7] [4]

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

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

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

Notation. *We shall assume the usual notation conventions on $\boldsymbol{\lambda}$-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 (Free variables)**.** *For every $\boldsymbol{\lambda}$-term M, we recursively define the set of free variables in M, $FV(M)$, as follows:*

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

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

Definition (α-equality)**.** *We say that two $\boldsymbol{\lambda}$-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 (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 (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:*

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

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

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

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

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

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

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

2.1.2 Types

[3]

Definition (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 (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 (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):*

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

2.2 Mechanising Meta-theory in Rocq

Having introduced the ordinary λ-calculus, we take it as an object of formalisation. This helps motivating the main decisions behind our mechanisations.

2.2.1 The Rocq Prover

(calculus of inductive constructions)

(Check Kathrin Stark introduction)

(Exemplo do tipo inductivo para inteiros?)

(Tipos mutuamente inductivos? Combined Schemes?)

2.2.2 Syntax with Binders

Formalising the λ-calculus in *Rocq* would generate 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 pen and paper proofs and definitions. 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.

The *Autosubst* library for the *Rocq Prover* stood out as a great solution for our case. This library uses a combination of de Bruijn indices and explicit parallel substitutions to ~~tackle this "problem"~~.

2.2.3 De Bruijn Syntax

[6] [8]

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

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

where i ranges over the natural numbers.

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

2.2.4 The Autosubst Library

[8]

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 $bindterm$ is as 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 On the Mechanisation of the 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 (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.
```

1. Contextos infinitos?

Chapter 3

The Multiary Lambda Calculus

3.1 The Multiary Lambda Calculus (λm)

Definition (**λm**-terms)**.** *The **λm**-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 (Concatenation)**.** *The concatenation of two **λm**-lists, $l + l'$, is recursively defined as follows:*

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

Definition (Substitution for **λm**-terms)**.** *The substitution over a **λm**-term is mutually defined with the substitution over a **λm**-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 (Reduction rules for **λm**-terms)**.**

$$(\lambda x.t)(u, []) \to t[x := u] \qquad (\beta_1)$$

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

$$t(u, l)(u', l') \to t(u, l + (u' :: l')) \qquad (h)$$

Definition (h-normal forms)**.** *We inductively define the sets of **λm**-terms (or canonical terms) and **λm**-lists in h-normal form, respectively Can and $CanList$, as follows:*

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

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

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

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

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

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

For this system, we still have the usual result of subject reduction.

Lemma 1 (Substitution Admissibility)**.**

Theorem 1 (Subject Reduction)**.**

3.2 The System $\vec{\lambda}$

Definition ($\vec{\lambda}$-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. *For every* $\vec{\lambda}$*-terms* t, u *and* $\vec{\lambda}$*-list* l*, we define the operation* $t@(u,l)$*, by the following equa-*

tions:

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

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

Definition (Substitution for $\vec{\boldsymbol{\lambda}}$-terms)**.** *The substitution over a $\vec{\boldsymbol{\lambda}}$-term is mutually defined with the substitution over a $\vec{\boldsymbol{\lambda}}$-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 (Typing Rules for $\vec{\boldsymbol{\lambda}}$-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 Formalised Results

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

$$\Gamma \vdash t : A \ \wedge\ t \to t' \implies \Gamma \vdash t' : A.$$

```
Theorem type_preservation :
  forall t t', step t t' → forall Γ A, sequent Γ t A → sequent Γ t' A.
Proof.
  intros t t' H.
  induction H using sim_comp_ind with (P0 := list_type_preservation) ;
    autounfold in * ; intros ;
    try (now inversion H ; econstructor ; eauto) ;
    try (now inversion H0 ; econstructor ; eauto).

  - inversion b.
    + inversion H0.
      * eapply beta1_type_preservation ; eassumption.
      * eapply beta2_type_preservation ; eassumption.
    + eapply h_type_preservation ; eassumption.
Qed.

Corollary type_preservation' Γ t t' A :
  sequent Γ t A ∧ step t t' → sequent Γ t' A.
Proof.
  intro H. destruct H as [H1 H2].
  eapply type_preservation ; eassumption.
Qed.
```

Theorem 3 (Conservativeness)**.** *Given $\vec{\lambda}\boldsymbol{m}$-terms t and t', we have the follwing:*

$$t \to^*_{\vec{\lambda}\boldsymbol{m}} t' \iff t \to^*_{\lambda\boldsymbol{m}} t'$$

```
Theorem conservativeness :
  forall t t', Canonical.multistep t t' ↔ LambdaM.multistep (i t) (i t').
Proof.
  split.
  - intro H.
    induction H as [| t1 t2 t3].
    + constructor.
    + apply multistep_trans with (i t2).
      * apply conservativeness1. assumption.
      * assumption.
```

```
- intro H.
  rewrite← inversion2 with t.
  rewrite← inversion2 with t'.

  induction H as [| t1 t2 t3].
  + constructor.
  + apply multistep_trans with (h t2).
    * apply conservativeness2. assumption.
    * assumption.
Qed.
```

3.4 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.

3.4.1 Mutually Inductive Type vs Nested Inductive Type

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 [8], 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. This implementation followed the approach explained in [8]. ...

At some point, the idea of using the polymorphic list type provided by the *Rocq* standard library came up.

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

This way, there was no need of having a mutual inductive type for our terms. After some further inspection of the library source code, we noticed that nested inductive types that depend on lists are already supported by default.

...falar da funcao mmap? ...

The downside of using nested inductive types in the *Rocq Prover* is the generated induction principles. This is 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 : forall x, P (Var x).
  Hypothesis HLam : forall t: {bind term}, P t → P (Lam t).
  Hypothesis HmApp : forall t u l, P t → P u → Q l → P (mApp t u l).
  Hypothesis HNil : Q [].
  Hypothesis HCons : forall u l, P u → Q l → Q (u::l).

  Proposition sim_term_ind : forall 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 (forall 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 : forall 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.
```

...

3.4.2 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 subsystem of canonical terms, that previously was presented as the system $\vec{\boldsymbol{\lambda m}}$, can be mechanised in many ways:

1. Use solely the predicate over $\boldsymbol{\lambda m}$-terms (for example, to declare that a property P is satisfied by every canonical term, one would have the proposition $\forall(t : \boldsymbol{\lambda m}), is\_canonical(t) \implies P(t)$);

2. Use the subset types provided by the standard library, that correspond to a pair of term t and a proof that t satisfies some predicate (in our case, t would satisfy the predicate of being canonical);

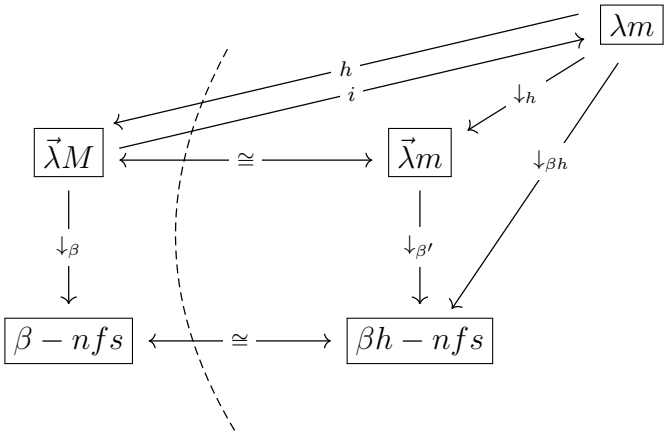
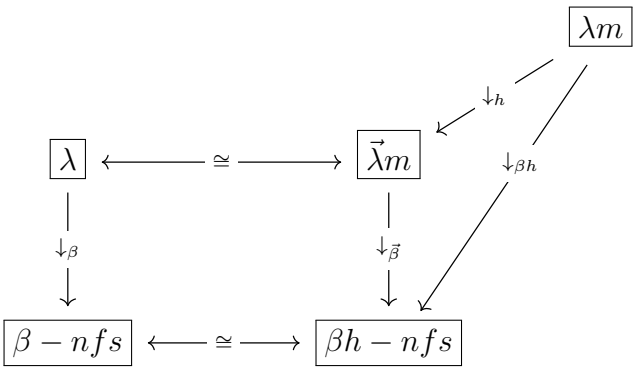


Figure 1: Systems formalised in *Rocq*

3. Have an isolated syntax (another inductive type) for these canonical terms.

Chapter 4

The Isomorphic Fragment of Lambda Calculus in λm



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

Chapter 6

Conclusions

Final chapter, present your conclusions.

6.1 Summary of Findings

Highlight per-chapter content here, with a general conclusion in the end.

6.2 Future Work

There's no lack of future work.

Appendix A

Example

This is what an Appendix looks like.

Bibliography

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

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

[8] 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.